MathSoft

S-PLUS 6.0 for UNIX User's Guide

October 2000

Data Analysis Division
MathSoft, Inc.
Seattle, Washington

Proprietary Notice

MathSoft, Inc. owns both this software program and its documentation. Both the program and documentation are copyrighted with all rights reserved by MathSoft.

The correct bibliographical reference for this document is as follows:

S-PLUS 6.0 User's Guide, Data Analysis Division, MathSoft, Seattle, WA.

Printed in the United States.

Copyright Notice

Copyright © 1987-2000, MathSoft, Inc. All rights reserved.

MathSoft, Inc.
101 Main Street
Cambridge, MA 02142
USA

Acknowledgments and Trademark Notices begin on page iii, which constitutes an extension of this Copyright Page.

Acknowledgments S-PLUS would not exist without the pioneering research of the Bell Labs S team at AT&T (now Lucent Technologies): Richard A. Becker (now at AT&T Laboratories), John M. Chambers, Allan R. Wilks (now at AT&T Laboratories), William S. Cleveland, Trevor Hastie (now at Stanford University), and colleagues.

S-PLUS owes a continuing debt to dozens of scientists and researchers who have contributed code to earlier releases. S-PLUS 6 includes new features contributed by a number of scientists:

The survival functions were written by Terry Therneau (Mayo Clinic, Rochester, Minnesota). The life testing functions include code contributed by W.Q. Meeker (Iowa State University).

The mixed-effects modeling functions were written by Doug Bates (University of Wisconsin–Madison) and José Pinheiro (Lucent Technologies).

The discriminant analysis function `discrim` contains code contributed by Brian Ripley (University of Oxford) and William Venables (CSIRO). The `digamma` and `trigamma` functions were written by William Venables (CSIRO).

Trademarks

- S-PLUS is a registered trademark, and StatServer, S+SDK, S+SPATIALSTATS, S+DOX, S+GARCH, S+WAVELETS, and Axum are trademarks of MathSoft, Inc.
- S and New S are trademarks of Lucent Technologies, Inc.
- Intel is a registered trademark, and 486, SX, and Pentium are trademarks of Intel Corporation
- Microsoft, Windows, MS-DOS, and Excel are registered trademarks, and Windows NT is a trademark of Microsoft Corporation.
- Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the U.S. and other countries.

Other brand and product names referred to are trademarks or registered trademarks of their respective owners.

CONTENTS

Chapter 1 Welcome to S-PLUS **1**

Introduction 2

Help, Support, and Learning Resources 3

Chapter 2 Getting Started **7**

Introduction 9

Running S-PLUS 10

Command Line Editing 18

Getting Help in S-PLUS 21

S-PLUS Language Basics 27

Importing and Editing Data 43

Graphics in S-PLUS 52

Statistics 57

Chapter 3 Working with the Graphical User Interface **63**

The User Interface 64

Using Menus, Dialog Boxes, and Toolbars 65

S-PLUS Windows 73

Chapter 4 Importing and Exporting Data 79
Introduction 80
Dialogs 81
Supported File Formats 93
Examples 95

Chapter 5 Data Frames 101
Introduction 102
The Benefits of Data Frames 103
Creating Data Frames 104
Combining Data Frames 109
Applying Functions to Subsets of a Data Frame 116
Adding New Classes of Variables to Data Frames 123

Chapter 6 Menu Graphics 125
Introduction 127
Scatter Plots 132
Visualizing One-Dimensional Data 157
Visualizing Two-Dimensional Data 174
Visualizing Three-Dimensional Data 183
Visualizing Multidimensional Data 191
Time Series 200
References 210

Chapter 7 Working With Graphics Devices 211
Printing Your Graphics 212
Graphics Window Details 231

Chapter 8 Statistics 261

Introduction 264
Summary Statistics 269
Compare Samples 276
Power and Sample Size 322
Experimental Design 327
Regression 334
Analysis of Variance 361
Mixed Effects 367
Generalized Least Squares 371
Survival 375
Tree 381
Compare Models 386
Cluster Analysis 389
Multivariate 401
Quality Control Charts 408
Resample 413
Smoothing 417
Time Series 421
References 428

Chapter 9 Customizing Your S-PLUS Session 429

Introduction 430
Setting S-PLUS Options 431
Setting Environment Variables 433
Customizing Your Session at Start-up and Closing 435
Using Personal Function Libraries 439

Specifying Your Working Directory 441
Specifying a Pager 442
Environment Variables and printgraph 443
Setting Up Your Window System 445

Appendix: Migrating From S-PLUS 3.4 449

Index 463

WELCOME TO S-PLUS 1

Introduction **2**

Help, Support, and Learning Resources **3**
Getting Help 3
Add-On Modules 3
StatLib 4
S-News 4
Training Courses 4
Technical Support 5
Books on Data Analysis Using S-PLUS 5

INTRODUCTION

Welcome to S-PLUS 6.0 for UNIX, the first release of S-PLUS for UNIX to include a Java-based graphical user interface (GUI) and extensive Java connectivity features.

As the exclusive licensee of the S language, MathSoft has molded the S technology into the most powerful data analysis product available today. The S-PLUS object-oriented environment delivers benefits that traditional language analysis programs simply can't match. With S-PLUS every data set, function, or analysis model is treated as an object, which makes it easy to examine and visually explore data, run functions one step at a time, and visually compare models for fit.

S-PLUS gives you immediate feedback because it runs functions one at a time. With S-PLUS, you've got control over every step of your analysis. Visually compare different models for fit, re-explore your data for outliers or other factors that might influence a result, and document every analysis function. Because S-PLUS puts you in control, you'll have complete confidence in the quality of your results.

When your analysis requires a new method or approach, you can modify existing methods or develop new ones with the programming language. By tapping into the power, flexibility and extensibility of S-PLUS, you can take your analysis to a new level.

HELP, SUPPORT, AND LEARNING RESOURCES

Getting Help

There are a variety of ways to accelerate your progress with S-PLUS, and to build upon the work of others. This section describes the learning and support resources available to S-PLUS users.

Online Help

S-PLUS offers an online help system to make learning and using S-PLUS easier. To get help, type `help.start()` at the S-PLUS prompt to start the new JavaHelp system.

Printed and Online Manuals

Your S-PLUS license comes with six manuals: this user's guide, a *Getting Started* guide, the two-volume *S-PLUS Guide to Statistics*, the *S-PLUS Programmer's Guide,* and the *S-PLUS Installation and Maintenance Guide*, all of which are also available online as PDF files.

Notes on Online versions of the Guides

The online manuals are viewed using Acrobat Reader, which is available for free over the Internet at **http://www.adobe.com** .

Add-On Modules

Add-on modules that offer analytical functionality beyond that of the base S-PLUS product include:

- **S+GARCH** : provides an essential suite of tools designed for univariate and multivariate GARCH modeling of financial time series data.
- **S+SPATIALSTATS**: provides a comprehensive set of tools for statistical analysis of spatial data, including tools for hexagonal binning, variogram estimation and kriging, autoregressive and moving average modeling, and testing for spatial randomness.
- **S+WAVELETS**: offers a visual data analysis approach to a whole range of signal-processing techniques, such as wavelet packets, local cosine analysis, and matching pursuits.

StatLib

StatLib is a system for distributing statistical software, data sets, and information by electronic mail, FTP and the World Wide Web. It contains a wealth of user-contributed S-PLUS functions.

- To access StatLib by FTP, open a connection to **lib.stat.cmu.edu**. Login as **anonymous** and send your e-mail address as your password. The FAQ (frequently asked questions) is in **/S/FAQ**, or in HTML format at **http://www.stat.math.ethz.ch/S-FAQ** .
- To access StatLib with a web browser, visit **http://lib.stat.cmu.edu/.**
- To access StatLib by e-mail, send the message: **send index from S** to **statlib@lib.stat.cmu.edu.** You can then request any item in StatLib with the request **send item from S** where **item** is the name of the item.

S-News

S-News is an electronic mailing list by which S-PLUS users can ask questions and share information with other users. To get on this list, send a message with message body **subscribe** to **s-news-request@wubios.wustl.edu**. To get off this list, send a message with body **unsubscribe** to the same address.

Once enrolled on the list, you will begin to receive e-mail. To send a message to the S-news mailing list, send it to: **s-news@wubios.wustl.edu**. Do *not* send subscription requests to the full list; use the s-news-request address shown above.

Training Courses

MathSoft Educational Services offers a variety of courses designed to quickly make you efficient and effective at analyzing data with S-PLUS. The courses are taught by professional statisticians and leaders in statistical fields. Courses feature a hands-on approach to learning, dividing class time between lecture and online exercises. All participants receive the educational materials used in the course, including lecture notes, supplementary materials, and exercise data on diskette.

Technical Support

To contact technical support in North America, call

(206) 283-8802 ext. 235
(800) 569-0123 ext. 235

or fax to

(206) 283-6310

or send e-mail to

support@statsci.com .

In Europe, Asia, Australia, Africa and South America, call

+44 1276 475350

or fax to

+44 1276 451224

or email to

shelp@mathsoft.co.uk.

Books on Data Analysis Using S-PLUS

General

Becker, R.A., Chambers, J.M., and Wilks, A.R. (1988). *The New S Language*. Wadsworth & Brooks/Cole, Pacific Grove, CA.

Chambers, J.M. (1998). *Programming with Data*. Springer-Verlag, New York.

Krause, A. and Olson, M. (1997). *The Basics of S and S-PLUS*. Springer-Verlag, New York.

Spector, P. (1994). *An Introduction to S and S-PLUS*. Duxbury Press, Belmont, CA.

Venables, W.N. and Ripley, B.D. (2000). *S Programming*. Springer-Verlag, New York.

Data Analysis

Bruce, A. and Gao, H.-Y. (1996). *Applied Wavelet Analysis with S-PLUS*. Springer-Verlag, New York.

Chambers, J.M., and Hastie, T.J. (1992). *Statistical Models in S*. Wadsworth & Brooks/Cole, Pacific Grove, CA.

Everitt, B. (1994). *A Handbook of Statistical Analyses Using S-PLUS*. Chapman & Hall, London.

Härdle, W. (1991). *Smoothing Techniques with Implementation in S*. Springer-Verlag, New York.

Kaluzny, S.P., Vega, S.C., Cardoso, T.P., and Shelly, A.A. (1997). *S+SPATIALSTATS User's Manual*. Springer-Verlag, New York.

Marazzi, A. (1992). *Algorithms, Routines and S Functions for Robust Statistics*. Wadsworth & Brooks/Cole, Pacific Grove, CA.

Pinheiro, J.C., and Bates, D.M. (2000). *Mixed-Effects Models in S and S-PLUS*. Springer-Verlag, New York.

Venables, W.N., and Ripley, B.D. (1999). *Modern Applied Statistics with S-PLUS*, Third Edition. Springer-Verlag, New York.

Graphical Techniques

Chambers, J.M., Cleveland, W.S., Kleiner, B., and Tukey, P.A. (1983). *Graphical Techniques for Data Analysis*. Duxbury Press, Belmont, CA.

Cleveland, W.S. (1993). *Visualizing Data*. Hobart Press, Summit, NJ.

Cleveland, W.S. (1985). *The Elements of Graphing Data*. Hobart Press, Summit, NJ.

GETTING STARTED 2

Introduction 9

Running S-PLUS 10
- Creating a Working Directory 10
- Starting S-PLUS 10
- Entering Expressions 14
- Quitting S-PLUS 15
- Basic Syntax and Conventions 16

Command Line Editing 18

Getting Help in S-PLUS 21
- Starting and Stopping the Help System 21
- Using the Help Window 21
- Getting Help at the S-PLUS Prompt 23
- Displaying Help in a Separate Window 25
- Printing Help Files 26
- Documentation Objects 26

S-PLUS Language Basics 27
- Data Objects 27
- Managing Data Objects 33
- Functions 35
- Operators 36
- Expressions 38
- Precedence Hierarchy 39
- Optional Arguments to Functions 41
- Access to UNIX 42

Importing and Editing Data **43**
Reading a Data File 43
Editing Data 44
Built-in Data Sets 45
Quick Hard Copy 46
Adding Row And Column Names 46
Extracting Subsets of Data 48

Graphics in S-PLUS **52**
Making Plots 52
Quick Hard Copy 55
Using the Graphics Window 55
Multiple Plot Layout 55

Statistics **57**
Summary Statistics 57
Hypothesis Testing 58
Statistical Models 59

INTRODUCTION

This chapter provides basic information that everyone needs to use S-PLUS effectively. It describes the following tasks:

- Starting and quitting S-PLUS
- Getting help
- Using fundamental elements of the S-PLUS language
- Creating and manipulating basic data objects
- Opening graphics windows and creating basic graphics

RUNNING S-PLUS

This section covers the basics of starting S-PLUS, opening windows for graphics and help, and constructing S-PLUS expressions.

Creating a Working Directory

Before running S-PLUS the first time, you should create a *working directory* specifically for S-PLUS. This directory will contain any files you want to read into or export from S-PLUS, as well as a **.Data** directory to hold your S-PLUS data objects, metadata objects, and help files. These working directories are called *chapters*, and are created with the S-PLUS **CHAPTER** utility. The first time you run S-PLUS, it creates a chapter called **MySwork** which can function as a default working directory; however, it will also store more general user information. MathSoft recommends creating at least one chapter separate from **MySwork**, and using that for your day-to-day S-PLUS work.

To create a working directory named **myproj** in your home directory, type the following sequence of commands at the UNIX shell prompt and press RETURN after each command:

```
cd
mkdir myproj
cd myproj
Splus CHAPTER
```

The **CHAPTER** utility creates a **.Data** directory, which in turn contains three other directories at start-up: **__Meta**, **__Shelp**, and **__Hhelp**. The **.Data** directory contains your normal data sets and functions, the **__Meta** directory contains S-PLUS metadata such as method definitions, and the two __***help** directories contain SGML and HTML versions of help files you create for your functions. All of these databases are initially empty, except for some possible marker files.

Starting S-PLUS

There are five basic ways to launch an S-PLUS session:

1. As a simple terminal-based application.
2. As a Java-controlled terminal-based application.

3. As a terminal-based application with command-line editing.
4. As a Java-based application with a graphical user interface.
5. As a batch operation.

S-PLUS as a Simple Terminal-Based Application

To start S-PLUS, type the following at the UNIX shell prompt and press the RETURN key:

```
Splus
```

Note that only the "S" is capitalized.

When you press RETURN, a copyright message appears in your S-PLUS window. The first time you that you start S-PLUS, you may also receive a message about initializing a new S-PLUS working directory. These messages are followed by the S-PLUS prompt:

```
Splus
S-PLUS : Copyright (c) 1988, 2000 MathSoft, Inc.
S : Copyright Lucent Technologies, Inc.
Version 6.0 for Sun SPARC, SunOS 5.5 : 2000
Working data will be in .
>
```

S-PLUS as a Java-Controlled Terminal-Based Application

To start S-PLUS as a terminal-based Java application, type the following at the UNIX shell prompt and press the RETURN key:

```
Splus -j
```

Note that only the "S" is capitalized.

When you press RETURN, a copyright message appears in your S-PLUS window. The first time you that you start S-PLUS, you may also receive a message about initializing a new S-PLUS working directory. These messages are followed by the S-PLUS prompt:

```
Splus -j
S-PLUS : Copyright (c) 1988, 2000 MathSoft, Inc.
S : Copyright Lucent Technologies, Inc.
Version 6.0 for Sun SPARC, SunOS 5.5 : 2000
Working data will be in .
>
```

S-PLUS as a Terminal-Based Application with Command-Line Editing

To start S-PLUS with command-line editing, add the **-e** flag to your normal start-up command. Thus, for the standard terminal-based S-PLUS, start the command-line editor as follows:

```
Splus -e
```

Note that only the "S" is capitalized.

For the Java-controlled terminal, start the command-line editor as follows:

```
Splus -j -e
```

When you press RETURN, a copyright message appears in your S-PLUS window. The first time you that you start S-PLUS, you may also receive a message about initializing a new S-PLUS working directory. These messages are followed by the S-PLUS prompt:

```
Splus -e
S-PLUS : Copyright (c) 1988, 2000 MathSoft, Inc.
S : Copyright Lucent Technologies, Inc.
Version 6.0 for Sun SPARC, SunOS 5.5 : 2000
Working data will be in .
>
```

For information on editing with the command-line editor, see the section Command Line Editing on page 18.

S-PLUS with a Graphical User Interface

To start S-PLUS with a graphical user interface, type the following at the UNIX shell prompt and press the RETURN key:

```
Splus -g &
```

Note that only the "S" is capitalized. The **&** indicates to the shell that the graphical user interface will run in the background; this simply allows the interface to start as a separate X window while returning the prompt to your UNIX shell window.

When you press RETURN, you will see the S-PLUS splash screen. Shortly thereafter, the graphical user interface appears, with menus, a toolbar, and a **Commands** window, as shown in Figure 2.1.

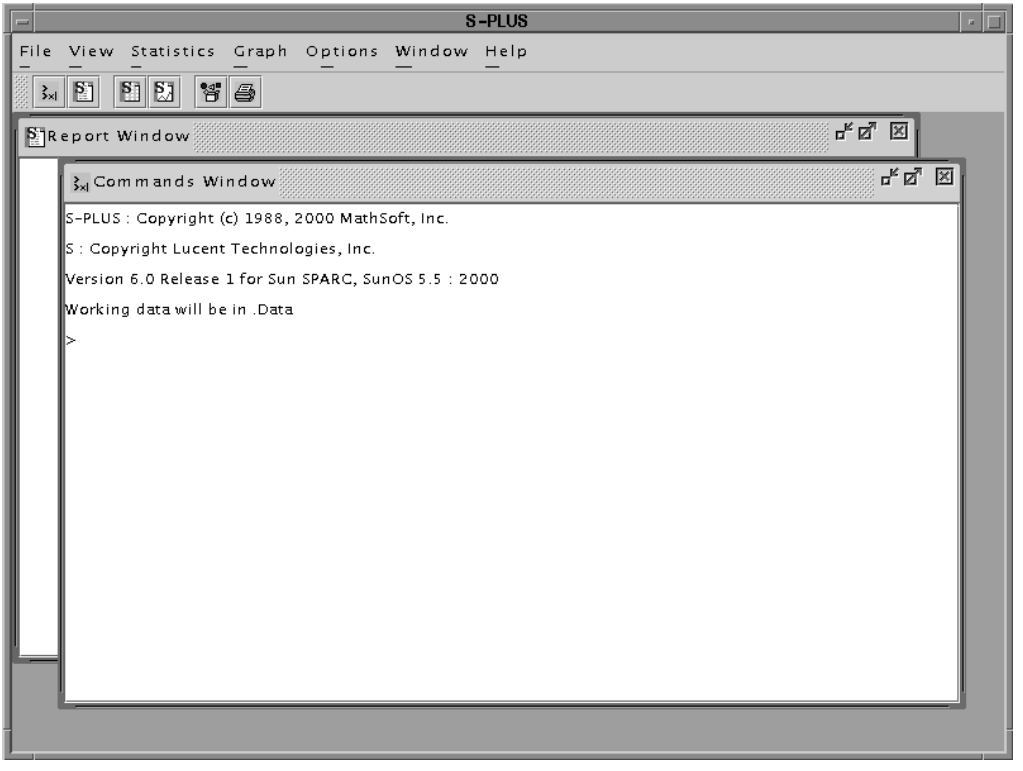

S-PLUS
File View Statistics Graph Options Window Help
Report Window
Commands Window
S-PLUS : Copyright (c) 1988, 2000 MathSoft, Inc.
S : Copyright Lucent Technologies, Inc.
Version 6.0 Release 1 for Sun SPARC, SunOS 5.5 : 2000
Working data will be in .Data
>

Figure 2.1: *The S-PLUS graphical user interface.*

A copyright message appears in the **Commands** window. The first time you that you start S-PLUS, you may also receive a message about initializing a new S-PLUS working directory. These messages are followed by the S-PLUS prompt:

```
S-PLUS : Copyright (c) 1988, 2000 MathSoft, Inc.
S : Copyright Lucent Technologies, Inc.
Version 6.0 for Sun SPARC, SunOS 5.5 : 2000
Working data will be in .
>
```

You can begin typing expressions in the **Commands** window, or you can use the menus and dialogs to perform S-PLUS tasks. Entering expressions is described in the section S-Plus as a Batch Process; using the menus and dialogs is introduced in the chapter Working with the Graphical User Interface.

S-PLUS as a Batch Process

Once you've created a function and verified that it works, you may want to use it with a large data set. Complicated analyses on large data sets can take some time, however, and your session is locked while S-PLUS performs its calculations. *Batch mode* provides one method for working around this. To run a set of commands in batch mode, simply create a file containing the S-PLUS expressions you want evaluated, and then type the following at the UNIX prompt:

```
Splus BATCH myfile myfile.out
```

Here, *myfile* is the name of the input file you create, and *myfile.out* is the name of the file in which S-PLUS should write the output.

When you run an S-PLUS process in batch mode, it begins immediately but is at a lower priority than interactive tasks. You can also run batch jobs from within an S-PLUS session by using the **!** shell escape:

```
> !Splus BATCH myfile myfile.out
```

Warning

When you run batch processes from within S-PLUS, the results are invisible to your current session; your working database is not updated with the results of the batch job. To see the results of a batch process in your current session, you must *synchronize* the databases. See the *Programmer's Guide* for more details.

Entering Expressions

You can use S-PLUS by typing expressions after the prompt and pressing the RETURN key. You type an expression at the S-PLUS > prompt, and S-PLUS responds.

Among the simplest S-PLUS expressions are arithmetic expressions such as the following:

```
> 3+7
[1] 10

> 3*21
[1] 63
```

The symbols "+" and "*" represent S-PLUS operators for addition and multiplication, respectively. In addition to the usual arithmetic and logical operators, S-PLUS has operators for special purposes. For example, the colon operator ":" is used to obtain sequences:

```
> 1:7
[1] 1 2 3 4 5 6 7
```

The `[1]` in each of the output lines is the *index* of the first S-PLUS response on the line of S-PLUS output. If S-PLUS is responding with a long vector of results, each line is preceded by the index of the first response of that line.

The most common S-PLUS expression is the *function call*. An example of a *function* in S-PLUS is `c`, which is used for "combining" comma-separated lists of items into a single object. Function calls are always followed by a pair of parentheses, with or without any *arguments* in the parentheses:

```
> c(3,4,1,6)
[1] 3 4 1 6
```

In all of our examples to this point, S-PLUS has simply returned a value. To reuse the value of an S-PLUS expression, you must *assign* it with the `<-` operator. For example, to assign the above expression to an S-PLUS object named `newvec`, type the following:

```
> newvec <- c(3, 4, 1, 6)
```

S-PLUS creates the object `newvec` and returns an S-PLUS prompt. To view the contents of the newly created object, just type its name:

```
> newvec
[1] 3 4 1 6
```

Quitting S-PLUS

To quit S-PLUS and get back to your UNIX shell prompt, use the `q` function:

```
> q()
```

The `()` are required with the `q` command to quit S-PLUS because `q` is an S-PLUS function, and parentheses are required with all S-PLUS functions. In the S-PLUS graphical user interface, you can also select **File ▶ Exit** to exit S-PLUS.

Basic Syntax and Conventions

This section introduces basic typing syntax and conventions in S-PLUS.

Spaces

S-PLUS ignores most spaces. For example:

```
> 3+     7
[1] 10
```

However, do not put spaces in the middle of numbers or names, or an error will result. For example, if you wish to add 321 and 1, the expression `32  1+1` causes an error. Also, you should always put spaces around the two-character assignment operator `<-`; otherwise, you may perform a comparison instead of an assignment.

Upper And Lower Case

S-PLUS is *case sensitive*, just like UNIX. All S-PLUS objects, arguments, and names are case sensitive. Hence, "QWERT" is different from "qwert". In the following example, the object `SeX` is defined as "M". You get an error message if you do not type "SeX" with the capitalization.

```
> SeX
[1] "M"

> sex
Problem: Object "sex" not found
```

Continuation

When you press the RETURN key and it is clear to S-PLUS that an expression is incomplete (for example, the last character is an operator, or there is a missing parenthesis), S-PLUS provides a *continuation* prompt to remind you to complete the expression. The default continuation prompt is "+".

Here are two examples of incomplete expressions that cause S-PLUS to respond with a continuation prompt:

```
> 3*
+ 21
[1] 63

> c(3,4,1,6
+)
[1] 3 4 1 6
```

In the first command, S-PLUS determined that the expression was not complete because the multiplication operator * must be followed by a data object. In the second example, S-PLUS determined that `c(3,4,1,6` was not complete because a right parenthesis is needed. In each of these cases, the user completed the expression after the continuation prompt (+), and then S-PLUS responded with the result of the complete evaluation.

Interrupting Evaluation Of An Expression

Sometimes you may want to stop the evaluation of an S-PLUS expression. For example, you may suddenly realize you want to use a different command, or the output display of data on the screen is extremely long and you don't want to look at all of it.

To interrupt S-PLUS from a terminal-based window, use the UNIX interrupt command, which consists of either CTRL-C (pressing the C key while holding down the CONTROL key) or the DELETE key on most systems. If neither CTRL-C nor DELETE stop the scrolling, consult your UNIX manual for use of the **stty** command to see what key performs the interrupt function, or consult your local system administrator.

To interrupt S-PLUS from the graphical user interface, press the ESC key on your keyboard.

Error Messages

Do not be afraid of making mistakes when using S-PLUS! You will not break anything by making a mistake. Usually you get some sort of error message, after which you can try again.

Here are two examples of mistakes made by typing "improper" expressions:

```
> 32 1+1
Problem: Syntax error: illegal literal ("1") on input line
1

> .5(2,4)
Problem: Invalid object supplied as function
```

In the second command, we typed something that S-PLUS tried to interpret as a function because of the parentheses. However, there is no function named `".5"`.

COMMAND LINE EDITING

Included with S-PLUS is a command line editor that can help improve your productivity. The S-PLUS command line editor enables you to recall and edit previously issued S-PLUS commands. The editor can do either **emacs**- or **vi**-style editing, and uses the first *valid* value in the following list of environment variables:

```
S_CLEDITOR
VISUAL
EDITOR
```

To be valid, the value for the environment variable must end in "vi" or "emacs." If none of the listed variables has a valid value, the command line editor defaults to **vi** style.

For example, issue the following command from the C shell to set your **S_CLEDITOR** to **emacs**:

```
setenv S_CLEDITOR emacs
```

To use the command line editor within S-PLUS, start S-PLUS with a **-e** option:

```
Splus -e
```

Table 2.1 summarizes the most useful editing commands for both **emacs**- and **vi**-style editing. With **vi**, the S-PLUS command line editor puts you in insert mode automatically. Thus, any editing commands must be preceded by an ESC.

Table 2.1: *Command line editing in S-PLUS.*

Action	emacs keystrokes	vi keystrokes*
backward character	CTRL-B	H
forward character	CTRL-F	L
previous line	CTRL-P	K
next line	CTRL-N	J

Table 2.1: *Command line editing in S-PLUS.*

Action	emacs keystrokes	vi keystrokes*
beginning of line	CTRL-A	SHIFT-6
end of line	CTRL-E	SHIFT-4
forward word	ESC,F	W
backward word	ESC,B	B
kill char	CTRL-D	X
kill line	CTRL-K	SHIFT-D
delete word	ESC,D	D,W
search backward	CTRL-R	/
yank	CTRL-Y	SHIFT-Y
transpose chars	CTRL-T	X,P
*In command mode. You must press ESC to enter command mode.		

As an example of using the command line editor, suppose you've started S-PLUS with the **emacs** option for the **EDITOR** environment variable. Attempt to create a plot by typing the following:

```
> plto(x,y)
Problem: Couldn't find a function definition for "plto"
```

Type CTRL-P to recall the previous line, then use CTRL-B to return to the "t" in "plto." Finally, type CTRL-T to transpose the "t" and the "o." Press RETURN to issue the edited command.

To recall earlier commands, use backward search (CTRL-R in **emacs** mode, / in **vi** mode) followed by the command or first portion of command. For example, suppose you've recently issued the following command:

```
> plot(xdata, ydata, xlab="Predictor", ylab="Response")
```

To recall this command, type CTRL-R `plot`. The complete command is restored to your command line. You can then use other editing commands to edit it if desired, or you can press RETURN to issue the command again.

GETTING HELP IN S-PLUS

If you need help at any time during an S-PLUS session, you can obtain it easily with the menu-driven help system, which uses Sun Microsystems' JavaHelp™. The S-PLUS window-driven help system lets you select from broad categories of help topics. Within each category you can choose from a list of S-PLUS functions pertaining to that category.

Starting and Stopping the Help System

The easiest way to access the help system is through the help window. To call up the help system, type `help.start()` at the > prompt. The `help.start` function no longer supports the `gui` argument, so don't type `help.start(gui="motif")` as you might have done in S-PLUS 3.4. A JavaHelp window appears, with a **Table of Contents** in the left pane. You will also see additional tabs for the **Index** and the **Search** capabilities.

To turn off the help system, type `help.off()` at the > prompt, and the JavaHelp window closes. To hide the help system temporarily, simply minimize or close the window (depending on your window manager).

In the S-PLUS graphical user interface, you can also select **Help ► Contents**, **Help ► Index**, or **Help ► Search** to view the help system's **Table of Contents**, **Index**, and **Search** lists, respectively. To close the GUI help window, click the **Close** button in the upper right corner of the interface. To turn the help system off, type `help.off()` in the **Commands** window.

Using the Help Window

The S-PLUS help window contains two panes. At start-up, the left-hand pane contains the **Table of Contents** while the right-hand pane is empty. The right pane is used to display help text. The left pane is tabbed, and contains pages for the help system's **Table of Contents**, **Index**, and **Search** lists. You can replace the **Table of Contents** with an **Index**, which is a listing of all the topics currently available, or with the **Search** pane, which allows you to perform a full-text search on the current help set.

Use the following steps to get help on a topic with the **Table of Contents**:

1. Scan the **Table of Contents** on the left side of the help window until you find the desired category. Use the scroll bars and the mouse buttons to scroll through the list.
2. To select the category, double-click on the category name, or single-click on the lever next to the folder icon for the category. Once you select a category, a list of S-PLUS functions and data sets pertaining to that category appears below the category name.
3. Scroll through the list of objects under the category name until you find the desired function.
4. To select the function, click on the function name. Once you select a function, S-PLUS formats the help file for that function and brings it up in the text pane.
5. Scroll through the help file using the scroll bars and the mouse buttons.
6. To print the formatted file, click the **Print** button on the JavaHelp toolbar.

Use the following steps to get help on a topic with the **Index**:

1. To select the help **Index**, click the middle tab in the left pane of the help window.
2. Move the pointer inside the **Find** text field.
3. Type the function name you wish to search for.
4. Press the RETURN key. In the text pane of the help window, S-PLUS displays the first help file in the **Index** list that matches the name of your function. To see help files for the remaining matches, continue to press the RETURN key.

Alternatively, you can scroll through the **Index** list until you find the function name that you want.

Use the following steps to get help on a topic with the full-text **Search**:

1. To select the help **Search**, click the right-most tab in the left pane of the help window.
2. Move the pointer inside the **Find** text field.
3. Type the word you wish to search for.
4. Press the RETURN key. A list of help topics matching your search criterion is displayed in the left pane. The topics are sorted in order of importance: the help files that contain your search criterion most often are displayed at the top of the list, along with the number of occurrences.
5. To select a function, double-click on the topic in the left pane of the help window. Once you select a topic, S-PLUS formats the help file for that function, brings it up in the text pane, and highlights your search criterion.

Getting Help at the S-PLUS Prompt

You can access help easily at the S-PLUS prompt with the `?` and `help` functions. The `?` function has simpler syntax and requires no parentheses in most instances:

```
> ?lm

(p1 of 6)

                    Fit Linear Regression Model

DESCRIPTION:

   Returns an object of class "lm" or "mlm" that represents
   a linear model fit.

USAGE:

lm(formula, data=<<see below>>, weights=<<see below>>,
    subset=<<see below>>, na.action=na.fail, method="qr",
    model=F, x=F, y=F, contrasts=NULL, ...)
```

```
REQUIRED ARGUMENTS:

   formula
          a formula object, with the response on the left of
          a `~' operator and the terms, separated by +
          operators, on the right. The response may be a
          single numeric variable or a matrix.

OPTIONAL ARGUMENTS:

   data
          data frame in which to interpret the variables
          named in the formula, subset, and weights
          arguments. This may also be a single number to
          handle some special cases -- see below for
          details. If data is missing, the variables in the
          model formula should be in the search path.
. . .
```

By default, both `?` and `help` use the **slynx** browser provided with S-PLUS to display the requested help; this is a version of the freely available **lynx** browser. If the JavaHelp system is running in your session, however, all subsequent requests for help files are sent to the help window. To view help files in **slynx** after the help system has been started, type `help.off()` at the S-PLUS prompt, and then request help with the `?` or `help` functions.

The text in the S-PLUS help files is formatted for display using HTML. You can use the arrow keys to page through a help file; use the "q" key to exit a help file and return to the S-PLUS prompt.

Note

By default, the JavaHelp system is launched when you start a GUI session of S-PLUS. If the help system is running in your session, type `help.off()` in the **Commands** window before using `?` and `help` to display files in the **slynx** browser.

You can specify a different help pager by using, for example, `options(help.pager="vi")`. Since **vi** is just a text editor, you will see all the HTML formatting codes if you use **vi** to view your help

files. One useful pager is `options(help.pager = "slynx -dump")`, which you can use to create formatted help files for viewing in other text editors.

The `?` command is particularly useful for obtaining information on classes of objects. If you use the syntax `class ?` with the name of a class, S-PLUS offers documentation on the class. For example,

```
> class ? timeSeries

                    Calendar Time Series Class

DESCRIPTION:

   The timeSeries class represents calendar time series
   objects in S-PLUS.

SLOTS:

   All of the slots except the last two, fiscal.year.start
   and type, are inherited from the base series class.

ARGUMENTS:
. . .
```

You can call `help` with the name of an S-PLUS function, operator, or data set as argument. For instance, the following command displays the help file for the `c` function:

```
> help("c")
```

The quotation marks are optional for most functions, but are required for functions and operators containing special characters, such as `<-`. Quotation marks are also required for S-PLUS reserved words, such as `for`, `in`, and `TRUE`.

Displaying Help in a Separate Window

The `help` function has an argument, `window=T`, that you can use to display your help files in a separate window from your S-PLUS session window. This allows you to view a help file while continuing to do work in your S-PLUS session. By default, the help window is a terminal window displaying the **slynx** browser, as determined by the setting of `options()$help.pager` . If you want to change your

browser settings, save the old options with the syntax `oldopts <- options(help.pager="whatever")`. To restore the **slynx** browser, call `options(oldopts)`.

The `window=T` argument applies only to terminal-based sessions of S-PLUS. In the graphical user interface, the `?` and `help` functions always display help files in a window that is separate from the **Commands** window. By default, the help window displays the **slynx** browser, as determined by the setting of `options()$help.pager`.

Printing Help Files

To print a help file, use the **Print** button in the JavaHelp window. For a more plainly formatted printed version, use the `help` function with the argument `offline=T`.

Documentation Objects

S-PLUS 5.1 and later do not support the *creation* of documentation objects, although you can still dump existing documentation objects and create help files used by the new S-PLUS help system. The `sourceDoc` function is now defunct.

S-PLUS LANGUAGE BASICS

This section introduces the most basic concepts you need to use the S-PLUS language: expressions, operators, assignments, data objects, and function calls.

Data Objects

When using S-PLUS, you should think of your data sets as *data objects* belonging to a certain *class*. Each class has a particular *representation*, often defined as a named list of *slots*. Each slot, in turn, contains an object of some other class. Among the most common classes are `"numeric"`, `"character"`, `"factor"`, `"list"`, and `"data.frame"`. This chapter introduces the most fundamental data objects; see the chapter Data Objects for a more detailed treatment.

The simplest type of data object is a one-way array of values, all of which are numbers, logical values, or character strings, but not a combination of those. For example, you can have an array of numbers: `-2.0 3.1 5.7 7.3`. Or you can have an array of logical values: `T T F T F T F F`, where `T` stands for `TRUE` and `F` stands for `FALSE`. Or you can have an ordered set of character strings: `"sharp claws"`, `"COLD PAWS"`. These simple one-way arrays are called *vectors* when stored in S-PLUS. The class `"vector"` is a *virtual class* encompassing all basic classes whose objects can be characterized as one-way arrays. In a vector, any individual value can be extracted and replaced by referring to its *index*, or position in the array. The *length* of a vector is the number of values in the array; valid indices for a vector object `x` are in the range `1:length(x)`. Most vectors belong to one of the following classes: `numeric`, `integer`, `logical`, or `character`. For example, the vectors described above have length 4, 8, and 2 and class `numeric`, `logical`, and `character`, respectively.

S-PLUS assigns the class of a vector containing different kinds of values in a way that preserves the maximum amount of information: character strings contain the most information, numbers contain somewhat less, and logical values contain still less. S-PLUS coerces less informative values to equivalent values of the more informative type:

```
> c(17, TRUE, FALSE)
[1] 17  1  0

> c(17, TRUE, "hello")
[1] "17"    "TRUE"  "hello"
```

Data Object Names

Object names must begin with a letter and may include any combinations of upper and lower case letters, numbers, and periods. For example, the following are all valid object names:

```
mydata
data.ozone
RandomNumbers
lottery.ohio.1.28.90
```

Warning

If you create S-PLUS data objects on a file system with more restrictive naming conventions than those your version of S-PLUS was compiled for, you may lose data if you violate the restrictive naming conventions. For example, if you are running S-PLUS on a machine allowing 255 character names and create S-PLUS objects on a machine restricting file names to 14 characters, object names greater than 14 characters will be truncated to the 14 character limit. If two objects share the same initial 14 characters, the latest object overwrites the earlier object. S-PLUS warns you whenever you attach a directory with more restrictive naming conventions than it is expecting.

Hint

You will not lose data if, when creating data objects on a file system with more restrictive naming conventions than your version of S-PLUS was compiled for, you restrict yourself to names that are unique under the more restrictive conventions. However, your file system may truncate or otherwise modify the object name. To recall the object, you must refer to it by its modified name. For example, if you create the object `aov.devel.small` on a file system with a 14 character limit, you should look for it in subsequent S-PLUS sessions with the 14 character name `aov.devel.smal`.

The use of periods often enhances the readability of similar data set names, as in the following:

```
data.1
data.2
data.3
```

Objects and methods created with S-PLUS 5.0 and later often follow a naming scheme that omits periods, but adds capital letters to enhance readability:

```
setMethod
signalSeries
```

Warning

You should not choose names that coincide with the names of S-PLUS functions. If you store a function with the same name as a built-in S-PLUS function, access to the S-PLUS function is temporarily prevented until you remove or rename the object you created. S-PLUS warns you when you have masked access to a function with a newly created function. To obtain a list of objects that mask other objects, use the `masked` function.

At least seven S-PLUS functions have single-character names: `C`, `D`, `c`, `I`, `q`, `s`, and `t`. You should be especially careful not to name one of your own functions `c` or `t`, as these are functions used frequently in S-PLUS.

Vector Data Objects

By now you are familiar with the most basic object in S-PLUS, the vector, which is a set of numbers, character values, logical values, etc. *Vectors must be of a single mode*: you cannot have a vector consisting of the values `T, -2.3`. If you try to create such a vector, S-PLUS coerces the elements to a common mode. For example:

```
> c(T,-2.3)
[1]  1.0 -2.3
```

Vectors are characterized by their *length* and *mode*. Length can be displayed with the `length` function, and mode can be displayed with the `mode` function.

Matrix Data Objects

An important data object type in S-PLUS is the *two-way array*, or *matrix* object. For example:

```
-3.0    2.1    7.6
 2.5    -.5   -2.6
 7.0   10.0   16.1
 5.3  -21.0   -6.5
```

Matrices and their higher-dimensional analogues, *arrays*, are related to vectors, but have an extra structure imposed on them. S-PLUS treats these objects similarly by having the matrix and array classes inherit from another virtual class, the `structure` class.

To create a matrix, use the `matrix` function. The `matrix` function takes as arguments a vector and two numbers which specify the number of rows and columns. For example:

```
> matrix(1:12, nrow=3, ncol=4)

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

In this example, the first argument to `matrix` is a vector of integers from 1 through 12. The second and third arguments are the number of rows and columns, respectively. Each row and column is labeled: the row labels are `[1,]`, `[2,]`, `[3,]` and the column labels are `[,1]`, `[,2]`, `[,3]`, `[,4]`. This notation for row and column numbers is derived from mathematical matrix notation.

In the above example, the vector `1:12` fills the first column first, then the second column, and so on. This is called filling the matrix "by columns." If you want to fill the matrix "by rows", use the optional argument `byrow=T` to `matrix`.

For a vector of given length used to fill the matrix, the number of rows determines the number of columns and vice versa. Thus, you need not provide both the number of rows and the number of columns as arguments to `matrix`; it is sufficient that you provide only one or the other. The following command produces the same matrix as above:

```
> matrix(1:12, 3)
```

You can also create this matrix by specifying the number of columns only. To do this, type:

```
> matrix(1:12, ncol=4)
```

You have to provide the optional argument `ncol=4` in *name=value* form because, by default, the second argument is taken to be the number of rows. When you use the "by name" form `ncol=4` as the second argument, you override the default. See the section Optional Arguments to Functions on page 41 for further information on using optional arguments in function calls.

The array classes generally have three slots: a `.Data` slot to hold the actual values, a `.Dim` slot to hold the dimensions vector, and an optional `.Dimnames` slot to hold the row and column names. The most important slot for a matrix data object is the dimension slot `.Dim`. You can use the `dim` function to display the dimensions of an object:

```
> my.mat <- matrix(1:8,4,2)
> dim(my.mat)
[1] 4 2
```

This shows that the dimension of the matrix `my.mat` is 4 rows by 2 columns. Matrix objects also have length and mode, which correspond to the length and mode of the vector in the `.Data` slot. You can use the `length` and `mode` functions to view these characteristics of a matrix. Like vectors, a matrix object has a single `mode`. This means that you cannot create, for example, a two column matrix with one column of numeric data and one column of character data. For that, you must use a data frame.

Data Frame Objects

S-PLUS contains an object called a *data frame* which is very similar to a matrix object. A data frame object consists of rows and columns of data, just like a matrix object, except that the columns can be of different modes. The following object, `baseball.df`, is a data frame consisting of some baseball data from the 1988 season. The first two columns are factor objects (codes for names of players), the next two columns are numeric, and the last column is logical.

```
> baseball.df

      bat.ID pitch.ID event.typ outs.play err.play
 r1 pettg001 clemr001         2         1        F
 r2 whitl001 clemr001        14         0        F
 r3 evand001 clemr001         3         1        F
 r4 trama001 clemr001         2         1        F
 r5 andeb001 morrj001         3         1        F
 r6 barrm001 morrj001         2         1        F
 r7 boggw001 morrj001        21         0        F
 r8 ricej001 morrj001         3         1        F
```

See the chapter Data Objects for further information on data frames. The chapter Importing and Exporting Data discusses how to read in data frame objects from ASCII files.

List Objects

The *list* object is the most general and most flexible object for holding data in S-PLUS. A list is an ordered collection of *components*. Each list component can be any data object, and different components can be of different modes. For example, a list might have three components consisting of a vector of character strings, a matrix of numbers, and another list. Hence, lists are more general than vectors or matrices because they can have components of different types or modes, and they are more general than data frames because they are not restricted to having a rectangular (row by column) nature.

You can create lists with the `list` function. To create a list with two components, one a vector of mode `"numeric"` and one a vector of character strings, type the following:

```
> list(101:119,c("char string 1","char string 2"))

[[1]]:
 [1] 101 102 103 104 105 106 107 108 109 110 111 112 113
[14] 114 115 116 117 118 119

[[2]]:
[1] "char string 1" "char string 2"
```

The components of the list are labeled by double square-bracketed numbers, here `[[1]]` and `[[2]]`. This notation distinguishes the numbering of list components from vector and matrix numbering. After each component label, S-PLUS displays the contents of that component.

For greater ease in referring to list components, it is often useful to name the components. You do this by giving each argument in the `list` function its own name. For instance, you can create the same list as above, but name the components "a" and "b" and save the list data object with the name `xyz`:

```
> xyz <- list(a = 101:119,
+ b = c("char string 1", "char string 2"))
```

To take advantage of the component names from the `list` command, use the name of the list, followed by a `$` sign, followed by the name of the component. For example, the following two commands display components `a` and `b`, respectively, of the list `xyz`.

```
> xyz$a
 [1] 101 102 103 104 105 106 107 108 109 110 111 112 113
[14] 114 115 116 117 118 119

> xyz$b
[1] "char string 1" "char string 2"
```

Managing Data Objects

In S-PLUS, any object you create at the command line is permanently stored on disk until you remove it. This section describes how to name, store, list, and remove your data objects.

Assigning Data Objects

To name and store data in S-PLUS, use one of the *assignment* operators `<-` or `=`. For example, to create a vector consisting of the numbers 4, 3, 2, and 1 and store it with the name `x`, use the `c` function as follows:

```
> x <- c(4,3,2,1)
```

You type `<-` by with two keys on your keyboard: the "less than" key (`<`) followed by the minus (`-`) character, with no intervening space.

To store the vector containing the integers 1 through 10 in `y`, type:

```
> y <- 1:10
```

The following assignment expressions use the operator `=`, are identical to the two assignments above:

```
> x = c(4,3,2,1)
> y=1:10
```

The `<-` form of the assignment operator is highly suggestive and readable, so the examples in this manual use the arrow. The `=` is easier to type and matches the assignment operator in C, so many users prefer it. However, the S language also uses the `=` operator inside function calls for argument matching; if you want assign the value of an argument inside a function call, you must use the `<-` operator.

Storing Data Objects

Data objects in your working directory are permanent. They remain even if you quit S-PLUS and start a new session later.

You can change the location where S-PLUS objects are stored by using the `attach` function. See the `attach` help file for further information. You can also change where your S-PLUS objects are located by explicitly specifying a new working directory. To do this, define the environment variable **S_WORK**, which can specify one directory or a colon-separated list of directories. The first valid directory in the list is used as your working directory. For more information on working directories, see the section Creating a Working Directory on page 10.

Listing Data Objects

To display a list of the data objects in your working directory, use the `objects` function as follows:

```
> objects()
```

If you created the vectors `x` and `y` in the section Assigning Data Objects on page 33, you see these listed in your working directory.

The S-PLUS function `objects` also searches for objects whose names match a character string given to it as an argument. The pattern search may include wildcard characters. For instance, the following expression displays all objects that start with the letter `d`:

```
> objects("d*")
```

For information on wildcards and how they work, see the help file for `grep`.

Removing Data Objects

Because S-PLUS objects are permanent, you should remove objects you no longer need from time to time. You can use the `rm` function to remove objects. The `rm` function takes any number of objects as its arguments, and removes each one from your working database. For instance, to remove two objects named `a` and `b`, use the following expression:

```
> rm(a,b)
```

Displaying Data Objects

To look at the contents of a stored data object, just type its name:

```
> x
[1] 4 3 2 1

> y
[1] 1 2 3 4 5 6 7 8 9 10
```

Functions

A *function* is an S-PLUS expression that returns a value, usually after performing some operation on one or more *arguments*. For example, the `c` function returns a vector formed by combining its arguments. You *call* a function by typing an expression consisting of the name of the function followed by a pair of parentheses, which may enclose some arguments separated by commas. For example, `runif` is a function which produces random numbers uniformly distributed between 0 and 1. To have S-PLUS compute 10 such numbers, type `runif(10)`:

```
> runif(10)
 [1] 0.6033770 0.4216952 0.7445955 0.9896273 0.6072029
 [6] 0.1293078 0.2624331 0.3428861 0.2866012 0.6368730
```

S-PLUS displays the results computed by the function, followed by a new prompt. In this case, the result is a vector object consisting of 10 random numbers generated by a uniform random number generator. The square-bracketed numbers, here `[1]` and `[6]`, help you keep track of how many numbers are displayed on each line of the output, and help you locate particular numbers.

One of the functions in S-PLUS that you will use frequently is the function `c` which allows you to combine data values into a vector. For example:

```
> c(3, 7, 100, 103)
[1] 3 7 100 103

> c(T, F, F, T, T)
[1] T F F F T T

> c("sharp teeth", "COLD PAWS")
[1] "sharp teeth" "COLD PAWS"

> c("sharp teeth", 'COLD PAWS')
[1] "sharp teeth" "COLD PAWS"
```

The last example illustrates that either double quotes (") or single quotes (') can be used to delimit character strings.

Usually, you want to assign the result of a function to an object with another name that is permanently saved (until you choose to remove it). For example:

```
> weather <- c("hot day","COLD NIGHT")
> weather
[1] "hot day" "COLD NIGHT"
```

Some functions in S-PLUS are commonly used with no arguments. For example, recall that you quit S-PLUS by typing `q()`. The parentheses are still required so that S-PLUS can recognize that the expression is a function.

When you leave the parentheses out of a function call, the function text is displayed on the screen. Typing any object's name causes S-PLUS to print that object; a function object is simply the definition of the function. To call the function, simply retype the function name with parentheses. For instance, if you accidentally type `q` instead of `q()` when you wish to quit S-PLUS, the body of the function `q` is displayed. In this case the body of the function is only two lines long:

```
> q
function(...)
.Internal(q(...), "S_dummy", T, 33)
>
```

No harm has been done. All you need to do now is correctly type `q()`, and S-PLUS returns to your UNIX system prompt.

```
> q()
%
```

Operators

An *operator* is a function that has at most two arguments, and can be represented by one or more special symbols which appear between the two arguments. For example, the usual arithmetic operations of addition, subtraction, multiplication and division are represented by the operators `+`, `-`, `*`, and `/`, respectively. Some simple calculations using the arithmetic operators are given in the examples below.

```
> 3+71
[1] 74

> 3*121
[1] 363

> (6.5 - 4)/5
[1] .5
```

The exponentiation operator is ^, which can be used as follows:

```
> 2 ^ 3
[1] 8
```

Some operators work with only one argument, and hence are called *unary* operators. For example, the subtraction operator - can act as a unary operator:

```
> -3
[1] -3
```

The colon (:) is an important operator for generating sequences of integers:

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
```

Table 2.2lists the S-PLUS operators for comparison and logic. Comparisons are among the most common sources for logical data:

```
> (1:10) > 5
 [1] F F F F F T T T T T
```

Comparisons and logical operations are frequently convenient for extracting subsets of data, and conditionals using logical comparisons play an important role in flow of control in functions.

Table 2.2: *Logical and comparison operators.*

Operator	Explanation	Operator	Explanation
==	equal to	!=	not equal to
>	greater than	<	less than
>=	greater than or equal to	<=	less than or equal to
&	vectorized And	|	vectorized Or
&&	control And	||	control Or
!	not		

Expressions

An *expression* is any combination of functions, operators, and data objects. Thus `x <- c(4,3,2,1)` is an expression that involves an operator (the assignment operator) and a function (the `c` function). Here are a few examples to give you an indication of the variety of expressions you will be using in S-PLUS:

```
> 3 * runif(10)
 [1] 1.6006757 2.2312820 0.8554818 2.4478138 2.3561580
 [6] 1.1359854 2.4615688 1.0220507 2.8043721 2.5683608

> 3*c(2,11)-1
[1]  5 32

> c(2*runif(5),10,20)
[1]  0.6010921  0.3322045  1.0886723  0.3510106
[5]  0.9838003 10.0000000 20.0000000

> 3*c(2*x,5)-1
[1] 41 14
```

The last two examples illustrate a general feature of S-PLUS functions: arguments to functions can themselves be S-PLUS expressions.

Here are three examples of expressions which are important because they show how arithmetic works in S-PLUS when you use expressions involving both vectors and numbers. If x consists of the numbers 4, 3, 2, and 1, then the following operations work on each element of x:

```
> x-1
[1] 3 2 1 0

> 2*(x-1)
[1] 6 4 2 0

> x ^ 2
[1] 16 9 4 1
```

Any time you use an operator with a vector as one argument and a number as the other argument, the operation is performed on each component of the vector.

Hint

If you are familiar with the APL programming language, this treatment of vectors will be familiar to you.

Precedence Hierarchy

The evaluation of S-PLUS expressions has a *precedence hierarchy*, shown in Table 2.3. Operators appearing higher in the table have higher precedence than those appearing lower; operators on the same line have equal precedence.

Among operators of equal precedence, evaluation proceeds from left to right within an expression. Whenever you are uncertain about the precedence hierarchy for evaluation of an expression, you should use parentheses to make the hierarchy explicit. S-PLUS shares a common feature of many computer languages that the innermost parentheses are evaluated first, and so on until the outermost parentheses are evaluated. In the following example, we assign the value 5 to a vector (of length 1) called x. We then use the *sequence* operator : and show the difference between how the expression is evaluated with and without parentheses.

Table 2.3: *Precedence of operators.*

Operator	Use
`$`	component selection
`[ [[`	subscripts, elements
`^`	exponentiation
`-`	unary minus
`:`	sequence operator
`%% %/% %*%`	modulus, integer divide, matrix multiply
`* /`	multiply, divide
`+ -`	add, subtract
`<> <= >= == !=`	comparison
`!`	not
`& | && ||`	and, or
`~`	formulas
`<<- -> <- _`	assignments

Note

When using the `^` operator, the exponent must be an integer if the base is a negative number.

For example, in the expression `1:(x-1)`, `(x-1)` is evaluated first, and S-PLUS displays the integers from 1 to 4 as a result:

```
> x <- 5
> 1:(x-1)

[1] 1 2 3 4
```

However, when the parentheses are left off, the `:` operator has greater precedence than the `-` operator. The expression `1:x-1` is interpreted by S-PLUS to mean "take the integers from 1 to 5, and then subtract one from each integer". Hence, the output is of length 5 instead of length 4, and starts at 0 instead of 1:

```
> 1:x-1
[1] 0 1 2 3 4
```

When using S-PLUS, keep in mind the effect of parentheses and the default operator hierarchy.

Optional Arguments to Functions

One powerful feature of S-PLUS functions is considerable flexibility through the use of *optional* arguments. At the same time, simplicity is maintained because sensible *defaults* for optional arguments have been built in, and the number of *required* arguments is kept to a minimum. You can determine which arguments are required and which are optional by looking in the help file under the REQUIRED ARGUMENTS and OPTIONAL ARGUMENTS sections.

For example, to produce 50 normal random numbers with mean 0 and standard deviation 1, use the following command:

```
> rnorm(50)
```

If you want to produce 50 normal random numbers with mean 3 and standard deviation 5, you can use any of the following:

```
> rnorm(50, 3, 5)
> rnorm(50, sd=5, mean=3)
> rnorm(50, m=3, s=5)
> rnorm(m=3, s=5, 50)
```

In the first expression, you supply the optional arguments *by value*. When supplying optional arguments by value, you must supply the arguments in the order they are given in the help file USAGE statement. In the second through fourth expressions, you supply the optional arguments *by name*. When supplying arguments by name, order is not important. However, we recommend that you supply optional arguments after required arguments for consistency of style. The third and fourth expressions above illustrate that you may abbreviate the formal names of optional arguments for convenience, so long as the abbreviations uniquely correspond to their respective argument names.

You will find that supplying arguments by name is convenient because you can supply them in any order. Of course, you do not need to specify *all* of the optional arguments. For instance, the following are two equivalent ways to produce 50 random normal numbers with mean 0 (the default), and standard deviation of 5:

```
> rnorm(50, m=0, s=5)
> rnorm(50, s=5)
```

Access to UNIX

One important feature of S-PLUS is easy access to and use of UNIX tools. S-PLUS provides a simple shell escape character for issuing a single UNIX command from within S-PLUS:

```
> !date
Mon Apr 15 17:46:25 PDT 1991
```

Here, `date` is a UNIX command which passes its result to S-PLUS for display as shown. You can use any UNIX command in place of `date`. Of course, if you have separate UNIX windows open on your workstation screen, you can simply move into another window to issue a UNIX command.

In addition to the escape function `!`, S-PLUS provides a `unix` function that is a more powerful way to execute UNIX commands. The `unix` function allows you to capture and manipulate output produced by UNIX within an S-PLUS session.

IMPORTING AND EDITING DATA

There are many kinds and sizes of data sets that you may want to work on in S-PLUS. The first step is to get your data into S-PLUS in appropriate data object form. In this section, we show you how to import data sets that exist as files and how to enter small data sets from your keyboard. For details on the **Import Data** dialog, see the chapter Importing and Exporting Data.

Reading a Data File

The data you are interested in may have been created in S-PLUS, but more likely it came to you in some other form. Perhaps your data is an ASCII file or is from someone else's work in another software package, such as SAS. You can read data from a variety of sources using the S-PLUS function `importData`.

For example, suppose you have a SAS file named **Exenvirn.ssd01** . To import that file using the `importData` function, you must supply the file's name as the `file` argument:

```
> Exenvirn <- importData(file = "Exenvirn.ssd01")
```

After S-PLUS reads the data file, it assigns the data to the `Exenvirn` data frame.

Entering Data From Your Keyboard

To get a small data set into S-PLUS, create an S-PLUS data object using the `scan()` function as follows:

```
> mydata <- scan()
```

where *mydata* is any legal data object name. S-PLUS prompts you for input, as described in the following example. We enter 14 data values and assign them to the object `diff.hs`. At the S-PLUS prompt, type in the name `diff.hs` and assign to it the results of the `scan` command. S-PLUS responds with the prompt `1:`, which means that you should enter the first value.

You can enter as many values per line as you like, separated by spaces. When you press RETURN, S-PLUS prompts with the index of the next value it is waiting for. In our example, S-PLUS responds with `6:` because you entered 5 values on the first line. When you finish entering data, press RETURN in response to the `:` prompt, and S-PLUS returns to the S-PLUS command prompt (>).

The complete example appears on your screen as follows:

```
> diff.hs <- scan()
1: .06 .13 .14 -.07 -.05
6: -.31 .12 .23 -.05 -.03
11: .62 .29 -.32 -.71
15:
>
```

Reading An ASCII File

Entering data from the keyboard is a relatively uncommon task in S-PLUS. More typically, you have a data set stored as an ASCII file that you want to read into S-PLUS. An ASCII file usually consists of numbers separated by spaces, tabs, newlines, or other delimiters.

Suppose you have a UNIX text file called **vec.data** in the same UNIX directory from which you started S-PLUS, and suppose **vec.data** contains the following data:

```
62 60 63 59
63 67 71 64 65 66
88 66 71 67 68 68
56 62 60 61 63 64 63 59
```

You read the **vec.data** file into S-PLUS by using the `scan` command with `"vec.data"` as an argument:

```
> x <- scan("vec.data")
```

The quotation marks around the `vec.data` argument to `scan` are required. You can now type `x` to display the data object you have read into S-PLUS.

If the UNIX file you want to read is not in the same directory from which you started S-PLUS, you must use the entire path name. If the UNIX text file **vec.data** is in a subdirectory with path name **/usr/mabel/test/vec.data** , then type:

```
> x <- scan ("/usr/mabel/test/vec.data")
```

Editing Data

After you have created an S-PLUS data object, you may want to change some of the data you have entered. The easiest way to modify simple vectors and S-PLUS functions is to use the `fix` function, which uses the editor specified in your S-PLUS session options. By default, the editor used is **vi**.

With `fix`, you create a copy of the original data object, edit it, then reassign the result under its original name. If you have a favorite editor, you can use it by specifying it with the `options` function. For example, if you prefer to use the **emacs** editor, you can set this up easily as follows:

```
> options(editor="emacs")
```

To create a *new* data object by modifying an existing object, use the `vi` function, assigning the result to a new name. For example, if you want to create your own version of a system function such as `lm`, you can use `vi` as follows:

```
> my.lm <- vi(lm)
```

Warning

If you do not assign the output from the `vi` function, the changes you make are simply scrolled across the screen, and are not incorporated into any function definition. The value is also stored in the object `.Last.value` until a new value is returned by S-PLUS. You can therefore recover the changes by *immediately* typing the following:

```
> myfunction <- .Last.value
```

Built-in Data Sets

S-PLUS comes with a large number of *built-in* data sets. These data sets provide examples for illustrating the capabilities of S-PLUS without requiring you to enter your own data. When S-PLUS is used as a teaching aid, the built-in data sets provide a foundation for problem assignments in data analysis.

To have S-PLUS display any of the built-in data sets, just type its name at the > prompt. The built-in data sets include data objects of various types, and are stored in a **data** directory of your search path. To see the databases that are attached to your search path by default, type `search()` at the S-PLUS command prompt:

```
> search()
[1] "MySwork"          "splus"            "stat"
[4] "data"             "trellis"          "nlme3"
[7] "main"
```

Your working directory is attached in the first position of your search path, and the **data** directory is attached in the fourth position. To see a listing of the built-in objects in the **data** directory, use the `objects` function as follows:

```
> objects("data")
 [1] "\001"           "..min.script.id"     ".Copyright"
 [4] ".Original"      ".PostScript.Options" ".Program"
 [7] ".Random.seed"       "CHAR"            "Defunct.funs"
[10] "Deprecated.funs"    "INT"             "LGL"
[13] "Lubricant"          "Puromycin"       "REAL"
[16] . . .
```

Quick Hard Copy

To obtain a quick hard copy of your S-PLUS objects, use the `lpr` function. For example, to print the object `diff.hs`, use the following command:

```
> lpr(diff.hs)
```

A copy of your data will be sent to your standard printer.

Adding Row And Column Names

Names can be added to a number of different types of S-PLUS objects. In this section we discuss adding labels to vectors and matrices.

Adding Names To Vectors

To add names to a vector of data, use the `names` function. You assign a character vector of length equal to the length of the data vector as the `names` attribute for the vector. For example, the following commands assign the integers 1 through 5 to a vector `x`, and assign the spelled out words for those integers to the `names` attribute of the vector:

```
> x <- 1:5
> names(x) <- c("one", "two", "three", "four", "five")
> x
 one two three four five
   1   2     3    4    5
```

You also use `names` to display the names associated with a vector:

```
> names(x)
 one two three four five
```

You should note that the class of simple data objects such as vectors may be changed when names are added. If a vector does not include names, S-PLUS recognizes it as a simple `"numeric"` object. When names are added, however, the class of the object changes to `"named"`:

```
> data.class(x)
[1] "named"
```

Adding Names To Matrices

In a matrix, both the rows and columns can be named. Often the columns have meaningful alphabetic word names because the columns represent different *variables*, while the row names are either integer values indicating the observation number or character strings identifying "case" labels. Lists are useful for adding row names and column names to a matrix, as we now illustrate.

The `dimnames` argument to the `matrix` function is used to name the rows and columns of the matrix. The `dimnames` argument must be a list with exactly 2 components. The first component gives the labels for the matrix rows, and the second component gives the names for the matrix columns. The length of the first component in the `dimnames` list is equal to the number of rows, and the length of the second component is equal to the number of columns.

For example, if we add a `dimnames` argument to the `matrix` command, the resulting matrix will have the row and column labels specified:

```
> matrix(1:12, nrow=3, dimnames=list(c('I','II','III'),
+ c('x1','x2','x3','x4')))

    x1 x2 x3 x4
  I  1  4  7 10
 II  2  5  8 11
III  3  6  9 12
```

You can assign row and column names to existing matrices using the `dimnames` function, which works much like the `names` function for vectors:

```
> y <- matrix(1:12, nrow=3)
> dimnames(y) <- list(c('I','II','III'),
+ c('x1','x2','x3','x4'))
```

```
> y
    x1 x2 x3 x4
  I  1  4  7 10
 II  2  5  8 11
III  3  6  9 12
```

Extracting Subsets of Data

Another powerful feature of the S-PLUS language is the ability to extract subsets of data for viewing or further manipulation. The examples in this section illustrate subset extraction for vectors and matrices only. However, similar techniques can be used to extract subsets of data from other S-PLUS data objects.

Subsetting From Vectors

Suppose you create a vector of length 5, consisting of the integers 5, 14, 8, 9, 5:

```
> x <- c(5, 14, 8, 9, 5)
> x
[1]  5 14  8  9  5
```

To display a single element of this vector, just type the vector's name followed by the element's index within square brackets. For example, type `x[1]` to display the first element and `x[4]` to display the fourth element:

```
> x[1]
[1] 5

> x[4]
[1] 9
```

To display more than one element at a time, use the `c` function within the square brackets. The following command displays the second and fifth elements of `x`:

```
> x[c(2,5)]
[1] 14  5
```

Use negation to display all elements *except* a a specified element or list of elements. For instance, `x[-4]` displays all elements except the fourth:

```
> x[-4]
[1]  5 14  8  5
```

Similarly, `x[-c(1,3)]` displays all elements except the first and third:

```
> x[-c(1,3)]
[1] 14  9  5
```

A more advanced use of subsetting uses a logical expression within the `[]` characters. Logical expressions divide a vector into two subsets: one for which a given condition is true, and one for which the condition is false. When used as a subscript, the expression returns the subset for which the condition is true.

For instance, the following expression selects all elements with values greater than 8:

```
> x[x>8]
[1] 14  9
```

In this case, the second and fourth elements of `x`, with values 14 and 9, meet the requirements of the logical expression `x > 8`, and are therefore displayed. As usual in S-PLUS, you can assign the result of the subsetting operation to another object. For example, you could assign the subset in the above expression to an object named `y`, and then display `y` or use it in subsequent calculations:

```
> y <- x[x>8]
> y
[1] 14  9
```

In the next section you will see that the same subsetting principles apply to matrix data objects, although the syntax is a little more complicated to account for both dimensions in a matrix.

Subsetting From Matrix Data Objects

A single element of a matrix can be selected by typing its coordinates inside the square brackets as an ordered pair, separated by commas. We use the built-in data set `state.x77` to illustrate. The first number inside the `[]` operator is the row index, and the second number is the column index. The following command displays the value in the third row, eighth column of `state.x77`:

```
> state.x77[3,8]
[1] 113417
```

You can also display an element, using row and column `dimnames`, if such labels have been defined. To display the above value, which happens to be in the row named "Arizona" and the column named "Area", use the following command:

```
> state.x77["Arizona", "Area"]
[1] 113417
```

To select sequential rows and/or columns from a matrix object, use the `:` operator. The following expression selects the first 4 rows and columns 3 through 5 and assigns the result to the object `x`:

```
> x <- state.x77[1:4, 3:5]
> x
           Illiteracy Life Exp Murder
 Alabama          2.1    69.05   15.1
  Alaska          1.5    69.31   11.3
 Arizona          1.8    70.55    7.8
Arkansas          1.9    70.66   10.1
```

The `c` function can be used to select non-sequential rows and/or columns of matrices, just as it was used for vectors. For instance, the following expression chooses rows 5, 22, and 44, and columns 1, 4, and 7 of `state.x77`:

```
> state.x77[c(5,22,44), c(1,4,7)]
           Population Life Exp Frost
California      21198    71.71    20
  Michigan       9111    70.63   125
      Utah       1203    72.90   137
```

As before, if row or column names have been defined, they can be used in place of the index numbers:

```
> state.x77[c("California","Michigan","Utah"),
+ c("Population","Life Exp","Frost")]

           Population Life Exp Frost
California      21198    71.71    20
  Michigan       9111    70.63   125
      Utah       1203    72.90   137
```

Selecting All Rows or All Columns From a Matrix Object

To select all of the rows in a matrix, leave the expression before the comma (in the square brackets) blank. To select all columns in a matrix, leave the expression after the comma blank. The following command chooses all columns in `state.x77` for the rows corresponding to California, Michigan, and Utah. In the expression, the closing bracket appears immediately after the comma; this means that all columns are selected.

```
> state.x77[c("California","Michigan","Utah"), ]
           Population Income Illiteracy Life Exp Murder
California      21198   5114        1.1    71.71   10.3
  Michigan       9111   4751        0.9    70.63   11.1
      Utah       1203   4022        0.6    72.90    4.5

           HS Grad Frost   Area
California    62.6    20 156361
  Michigan    52.8   125  56817
      Utah    67.3   137  82096
```

GRAPHICS IN S-PLUS

Graphics are central to the S-PLUS philosophy of looking at your data visually as a first and last step in any data analysis. With its broad range of built-in graphics functions and its programmability, S-PLUS lets you look at your data from many angles. This section describes how to use S-PLUS to create simple command-line plots. To put S-PLUS to work creating the many other types of plots, see the chapters Traditional Graphics and Traditional Trellis Graphics.

This section is geared specifically to graphics that are created by S-PLUS functions and displayed in `motif` windows. For information on manipulating **Graph** windows in the GUI, see the chapter Working with the Graphical User Interface. For information on creating plots from the **Graph** menu options in the GUI, see the chapter Menu Graphics.

Making Plots

Plotting engineering, scientific, financial or marketing data, including the preparation of camera-ready copy on a laser printer, is one of the most powerful and frequently used features of S-PLUS. S-PLUS has a wide variety of plotting and graphics functions for you to use.

The most frequently used S-PLUS plotting function is `plot`. When you call a plotting function, an S-PLUS graphics window displays the requested plot:

```
> plot(car.miles)
```

The argument `car.miles` is an S-PLUS built-in vector data object. Since there is no other argument to `plot`, the data are plotted against their natural index or observation numbers, 1 through 120. Since you may be interested in gas mileage, you can plot `car.miles` against `car.gals`. This is also easy to do with `plot`:

```
> plot(car.gals, car.miles)
```

The result is shown in Figure 2.2.

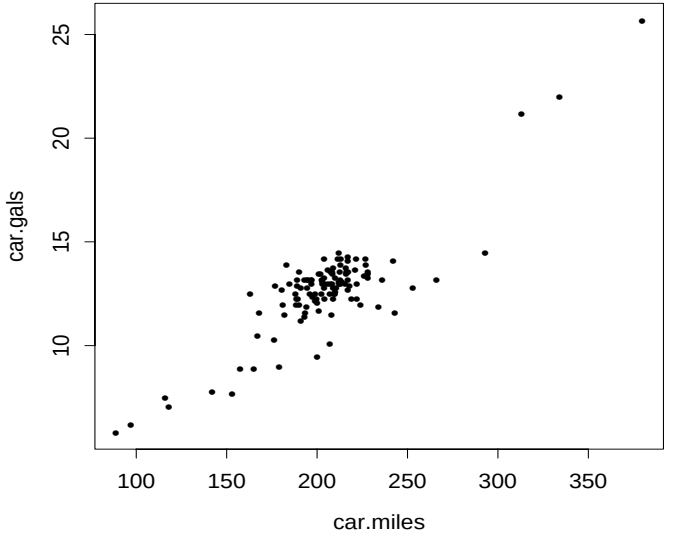


Figure 2.2: *An S-PLUS plot.*

You can use many S-PLUS functions besides `plot` to display graphical results in the S-PLUS graphics window. Many of these functions are listed in Table 2.4 and Table 2.5, which display, respectively, high-level and low-level plotting functions. High-level plotting functions create new plots and axes, while low-level plotting functions typically add to an existing plot.

Table 2.4: *Common high-level plotting functions.*

`barplot, hist`	Bar graph, histogram
`boxplot`	Boxplot
`brush`	Brush pair-wise scatter plots; spin 3D axes
`contour, image, persp, symbols`	3D plots
`coplot`	Conditioning plot

Table 2.4: *Common high-level plotting functions. (Continued)*

`dotchart`	Dot chart
`faces, stars`	Display multivariate data
`map`	Plot all or part of the U.S. (this function is part of the `maps` library)
`pairs`	Plot all pair-wise scatter plots
`pie`	Pie chart
`plot`	Generic plotting
`qqnorm, qqplot`	Normal and general QQ-plots
`scatter.smooth`	Scatter plot with a smooth curve
`tsplot`	Plot a time series
`usa`	Plot the boundary of the U.S.

Table 2.5: *Common low-level plotting functions.*

`abline`	Add line in intercept-slope form
`axis`	Add axis
`box`	Add a box around plot
`contour, image, persp, symbols`	Add 3D information to plot
`identify`	Use mouse to identify points on a graph
`legend`	Add a legend to the plot
`lines, points`	Add lines or points to a plot
`mtext, text`	Add text in the margin or in the plot
`stamp`	Add date and time information to the plot
`title`	Add title, *x*-axis labels, *y*-axis labels, and/or subtitle to plot

Quick Hard Copy

Each graphics window offers a simple, straightforward way to obtain a hard copy of the picture you have composed on the screen: the **Print** option under the **Graph** pull-down menu. You can exercise more control over your instant hard copy, by specifying whether the copy is in landscape or portrait orientation, which printer the hard copy is sent to, and for HP-Laserjet systems, the dpi (dots per inch) resolution of the printout.

Using the Graphics Window

You can use a mouse to perform basic functions in a graphics window, such as redrawing or copying a graph. The standard graphics window, also known as the `motif` device (Figure 2.3) has a set of pull-down menus providing a mouse-based point and click capability for copying, redrawing and printing hard copy on a printer.

In general, you select actions by pulling down the appropriate menu, and clicking the left mouse button.

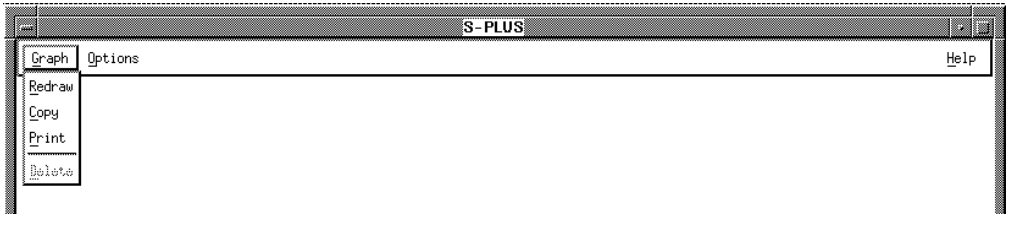


Figure 2.3: *The* `motif` *window.*

Copying A Graph

Each graphics window provides a mechanism to copy a graph on the screen. This option allows you to "freeze" a picture in one state, but continue to modify the original. The `motif` device has a **Copy** choice under the **Graph** pull-down menu.

Redrawing A Graph

Each graphics window provides a mechanism for "redrawing" a graph. This option can be used to refresh the picture if your screen has become cluttered. The `motif` device offers the **Redraw** option as a selection from the **Graph** pull-down menu.

Multiple Plot Layout

It is often desirable to display more than one plot in a window or on a single page of hard copy. To do so, you use the S-PLUS function `par` to control the layout of the plots. The following example shows how to

use par for this purpose. The par command is used to control and customize many aspects of S-PLUS plots. See the chapter Traditional Graphics for further information on the par command.

In this example, we use par to set up a a window or a page that has four plots in two rows of two each. Following the par command, we issue four plotting commands. Each command creates a simple plot with a main title.

```
> par(mfrow=c(2,2))
> plot(1:10,1:10,main="Straight Line")
> hist(rnorm(50),main="Histogram of Normal")
> qqnorm(rt(100,5),main="Samples from t(5)")
> plot(density(rnorm(50)),main="Normal Density")
```

The result is shown in Figure 2.4.

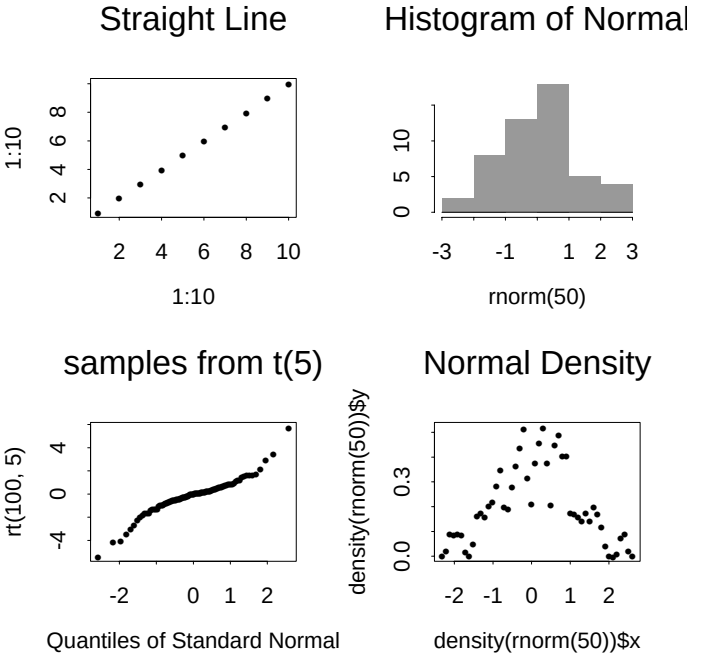


Figure 2.4: *A multiple plot layout.*

STATISTICS

S-PLUS includes functions for doing all kinds of statistical analysis, including hypothesis testing, linear regression, analysis of variance, contingency tables, factor analysis, survival analysis, and time series analysis. Estimation techniques for all these branches of statistics are described in detail in the manual *S-PLUS Guide to Statistics*.

This section gives overviews of the functions that produce summary statistics, perform hypothesis tests, and fit statistical models. This section is geared specifically to statistical analyses that are generated by S-PLUS command-line functions. For information on the options available under the **Statistics** menu in the GUI, see the Statistics chapter.

Summary Statistics

S-PLUS includes functions for calculating all of the standard summary statistics for a data set, together with a variety of robust and/or resistant estimators of location and scale. Table 2.6 lists of the most common functions for summary statistics.

The `summary` function is a generic function that provides appropriate summaries for different types of data. For example, an object of class `lm` created by fitting a linear model has a summary that includes the table of estimated coefficients, their standard errors, and t-values, along with other information. The summary for a standard vector is a six-number table of the minimum, maximum, mean, median, and first and third quartiles:

```
> summary(stack.loss)
 Min. 1st Qu. Median  Mean 3rd Qu. Max.
    7      11     15 17.52      19   42
```

Table 2.6: *Common functions for summary statistics.*

`cor`	Correlation coefficient
`cummax, cummin, cumprod, cumsum`	Cumulative maximum, minimum, product, and sum
`diff`	Create sequential differences
`max, min`	Maximum and minimum

Table 2.6: *Common functions for summary statistics. (Continued)*

`pmax, pmin`	Maxima and minima of several vectors
`mean`	Arithmetic mean
`median`	50th percentile
`prod`	Product of elements of a vector
`quantile`	Compute empirical quantiles
`range`	Returns minimum and maximum of a vector
`sample`	Random sample or permutation of a vector
`sum`	Sum elements of a vector
`summary`	Summarize an object
`var`	Variance and covariance

Hypothesis Testing

S-PLUS contains a number of functions for doing classical hypothesis testing, as shown in Table 2.7. The following example illustrates how to use `t.test` to perform a two-sample t-test to detect a difference in means. This example uses two random samples generated from N(0,1) and N(1,1) distributions. We set the random number seed with the function `set.seed` so this example is reproducible:

```
> set.seed(19)
> x <- rnorm(10)
> y <- rnorm(5, mean=1)
> t.test(x,y)
  Standard Two-Sample t-Test

data:  x and y
t = -1.4312, df = 13, p-value = 0.176
alternative hypothesis: true difference in means is not
  equal to 0
95 percent confidence interval:
 -1.7254080  0.3502894
sample estimates:
  mean of x mean of y
 -0.4269014 0.2606579
```

Table 2.7: *S-PLUS functions for hypothesis testing.*

Test	Description
`t.test`	Student's one- or two-sample t-test
`wilcox.test`	Wilcoxon rank sum and signed-rank sum tests
`chisq.test`	Pearson's chi square test for 2D contingency table
`var.test`	F test to compare two variances
`kruskal.test`	Kruskal-Wallis rank sum test
`fisher.test`	Fisher's exact test for 2D contingency table
`binom.test`	Exact binomial test
`friedman.test`	Friedman rank sum test
`mcnemar.test`	McNemar's chi square test
`prop.test`	Proportions test
`cor.test`	Test for zero correlation
`mantelhaen.test`	Mantel-Haenszel chi square test

Statistical Models

Most of the statistical modeling functions in S-PLUS follow a unified modeling paradigm in which the input data are represented as a data frame and the model to be fit is represented as a formula. Formulas can be saved as separate S-PLUS objects and supplied as arguments to the modeling functions. A partial listing of S-PLUS modeling functions is given in Table 2.8.

In a formula, you specify the response variable first, followed by a tilde (~) and the terms to be included in the model. Variables in formulas can be any expression that evaluates to a numeric vector, a factor or ordered factor, or a matrix. Table 2.9 gives a summary of the formula syntax.

Table 2.8: *S-PLUS modeling functions.*

Function	Description
`aov, manova`	Analysis of variance models
`lm`	Linear model (regression)
`glm`	Generalized linear model (including logistic and Poisson regression)
`gam`	Generalized additive model
`loess`	Local regression model
`tree`	Classification and regression tree models
`nls, ms`	Nonlinear models
`lme, nlme`	Mixed-effects models
`factanal`	Factor analysis
`princomp`	Principal components analysis
`pam, fanny, diana, agnes, daisy, clara`	Cluster analysis

Table 2.9: *Summary of the S-PLUS formula syntax.*

Expression	Meaning
`A ~ B`	`A` is modeled as `B`
`B + C`	Include both `B` and `C` in the model
`B - C`	Include all of `B` except what is in `C` in the model
`B:C`	The interaction between `B` and `C`
`B*C`	Include `B`, `C`, and their interaction in the model
`C %in% B`	`C` is nested within `B`
`B/C`	Include `B` and `C %in% B` in the model

The following sample S-PLUS session illustrates some steps to fit a regression model to the `fuel.frame` data containing five variables for 60 cars. We do not show the output; type these commands at your S-PLUS prompt and you'll get a good feel for doing data analysis with the S-PLUS language.

```
> names(fuel.frame)
> par(mfrow=c(3,2))
> plot(fuel.frame)
> pairs(fuel.frame)
> attach(fuel.frame)
> par(mfrow=c(2,1))
> scatter.smooth(Mileage ~ Weight)
> scatter.smooth(Fuel ~ Weight)
> lm.fit1 <- lm(Fuel ~ Weight)
> lm.fit1
> names(lm.fit1)
> summary(lm.fit1)
> qqnorm(residuals(lm.fit1))
> plot(lm.influence(lm.fit1)$hat, type="h",
+    xlab = "Case Number", ylab = "Hat Matrix Diagonal")
> o.type <- ordered(Type, c("Small", "Sporty", "Compact",
+    "Medium", "Large", "Van"))
> par(mfrow=c(1,1))
> coplot(Fuel ~ Weight | o.type,
+    given.values=sort(unique(o.type)))
> lm.fit2 <- update(lm.fit1, . ~ . + Type)
> lm.fit3 <- update(lm.fit2, . ~ . + Weight:Type)
> anova(lm.fit1, lm.fit2, lm.fit3)
> summary(lm.fit3)
```

WORKING WITH THE GRAPHICAL USER INTERFACE 3

The User Interface 64

Using Menus, Dialog Boxes, and Toolbars 65
- Using the Mouse 65
- Using the Keyboard 66
- Using Windows 66
- Using Main Menus 70
- Specifying Options in Dialogs 70
- Using Toolbar Buttons 72

S-PLUS Windows 73
- **Objects Summary** 73
- **Data Viewer** 73
- **Graph** Window 74
- **Commands** Window 75
- **Report** Window 75
- S-PLUS Menus 76
- S-PLUS Dialogs 77

THE USER INTERFACE

S-PLUS is a full-featured statistics and graphics application designed for easy, intuitive analysis and visualization of data. The Java-based graphical user interface makes this work even easier. This chapter gives an overview of the menus, windows, and toolbars that are the backbone of the product.

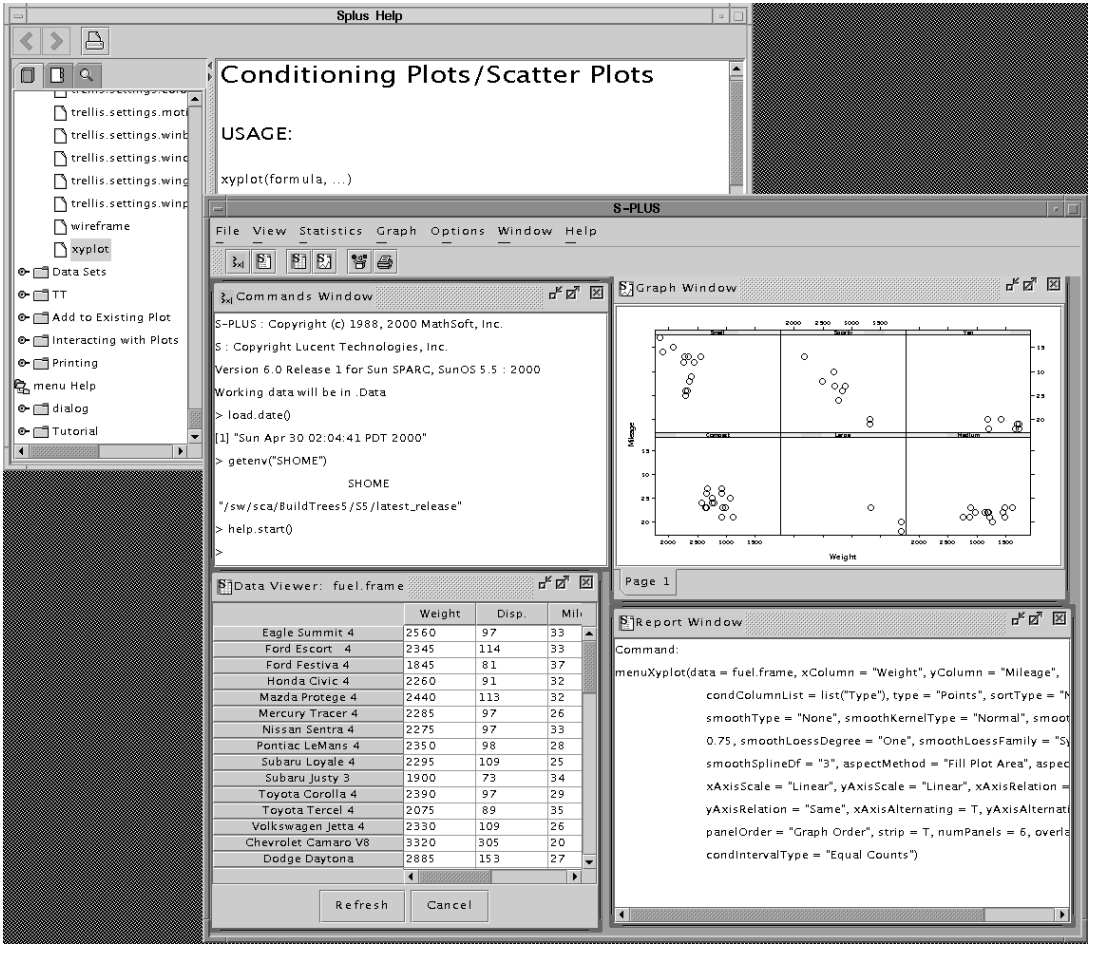

Splus Help
Conditioning Plots/Scatter Plots
USAGE:
xyplot(formula, ...)
trellis.settings.mot
trellis.settings.winb
trellis.settings.winc
trellis.settings.wing
trellis.settings.winp
wireframe
xyplot
Data Sets
TT
Add to Existing Plot
Interacting with Plots
Printing
menu Help
dialog
Tutorial
S-PLUS
File View Statistics Graph Options Window Help
Commands Window
S-PLUS : Copyright (c) 1988, 2000 MathSoft, Inc.
S : Copyright Lucent Technologies, Inc.
Version 6.0 Release 1 for Sun SPARC, SunOS 5.5 : 2000
Working data will be in .Data
> load.date()
[1] "Sun Apr 30 02:04:41 PDT 2000"
> getenv("SHOME")
SHOME
"/sw/sca/BuildTrees5/S5/latest_release"
> help.start()
Graph Window
Mileage
Weight
Page 1
Data Viewer: fuel.frame
Weight Disp. Mil
Eagle Summit 4 2560 97 33
Ford Escort 4 2345 114 33
Ford Festiva 4 1845 81 37
Honda Civic 4 2260 91 32
Mazda Protege 4 2440 113 32
Mercury Tracer 4 2285 97 26
Nissan Sentra 4 2275 97 33
Pontiac LeMans 4 2350 98 28
Subaru Loyale 4 2295 109 25
Subaru Justy 3 1900 73 34
Toyota Corolla 4 2390 97 29
Toyota Tercel 4 2075 89 35
Volkswagen Jetta 4 2330 109 26
Chevrolet Camaro V8 3320 305 20
Dodge Daytona 2885 153 27
Refresh
Cancel
Report Window
Command:
menuXyplot(data = fuel.frame, xColumn = "Weight", yColumn = "Mileage",
condColumnList = list("Type"), type = "Points", sortType = "
smoothType = "None", smoothKernelType = "Normal", smoot
0.75, smoothLoessDegree = "One", smoothLoessFamily = "Sy
smoothSplineDf = "3", aspectMethod = "Fill Plot Area", aspec
xAxisScale = "Linear", yAxisScale = "Linear", xAxisRelation =
yAxisRelation = "Same", xAxisAlternating = T, yAxisAlternat
panelOrder = "Graph Order", strip = T, numPanels = 6, overla
condIntervalType = "Equal Counts")

Figure 3.1: *S-PLUS in action; showing both the JavaHelp window (top left) and the S-PLUS graphical user interface (below right). Within the S-PLUS window, note the* ***Graph*** *window (top right), the* ***Data Viewer*** *(below left), and* ***Report*** *window (below right).*

USING MENUS, DIALOG BOXES, AND TOOLBARS

S-PLUS menus, dialogs and toolbars contain all the options you need to view data, create graphs, and perform statistical analyses. You can use your mouse or your keyboard to access S-PLUS's menus. Dialogs can be accessed by selecting menu options. Mouse, keyboard and window terms used throughout this document are defined below.

Using the Mouse

Throughout this document, the following conventions are used to describe mouse operations.

- **Pointing**: moving the mouse to position the pointer over an object.
- **Clicking**: pointing at an object and quickly pressing and releasing the left mouse button. Some tasks in S-PLUS require a *double-click*, which is achieved by quickly pressing and releasing the left mouse button twice.
- **Right-Clicking**: pointing at a selected object and quickly pressing and releasing the right mouse button.
- **Dragging**: pointing at the object, then holding down the left mouse button while moving the mouse. Releasing the left mouse button "drops" the object in the new location.

The mouse pointer changes shape to indicate what action is taking place. The following table shows the different mouse pointer shapes and the significance of each.

Table 3.1: *Different shapes of the mouse pointer.*

Pointer	Mouse Action
	Selection mouse pointer.
	Text indicator, slanted pointer indicates italic text.
	Displayed when Move or Size is selected from the Control menu; allows the window to be moved or resized.
	Change the size of the window vertically or horizontally when positioned on a window border.
	Change the size of two sides of the window when positioned on the corner of a window border.
	Indicates that a command is being processed; you should wait for a different mouse pointer before going on to other tasks.

Using the Keyboard

Throughout this document, the following conventions are used to reference keys.

- Key names appear in SMALLCAPS letters. For example, the Shift key appears as SHIFT.
- When more than one key must be pressed simultaneously, the two key names appear with a plus (+) between them. For example, the key combination of SHIFT and F1 appears as SHIFT+F1.
- The up, down, left, and right direction keys (represented on the keyboard by arrows) are useful for moving objects around the page. They are referred to as the UP direction key, the DOWN direction key, the LEFT direction key, and the RIGHT direction key.

Using Windows

In S-PLUS you can operate on multiple windows, making it easy to view different data sets and display multiple graphs. The graphical user interface is contained within a single main window, and has multiple subwindows.

The Control-menu box is always in the upper-left corner of the main S-PLUS window. Click once on the Control-menu box for a list of commands that control the size, shape, and attributes of the window. Click twice on the Control-menu box to quit S-PLUS.

The title bar displays the name of the window. If more than one window is open, the title bar of the current (or active) window is a different color or intensity than other title bars.

The Minimize button is represented in the main S-PLUS window by a small box, and in the subwindows by a small box with an arrow pointing into it. When this button is clicked, the window is reduced to an icon.

The Maximize button is represented in the main S-PLUS window by a large box, and in the subwindows by a large box with an arrow pointing out of it. When this button is clicked, the main S-PLUS window enlarges to fill the entire desktop, or the subwindow enlarges to fill the entire S-PLUS window.

The Restore button replaces the Maximize button when the window is maximized. The Restore button contains a large square with an arrow pointing into it, and it returns the window to its previous size.

The Close button is available only in the subwindows, and is not included as part of the main S-PLUS window. The Close button is represented by a square with an "X" in it, and it is used to close the **Commands** window, the **Report** window, **Graph** windows, etc.

The menu bar is a list of the available menus. Each menu contains a list of commands or actions.

The scroll bars let you scroll up and down through a window.

The window border surrounds the entire window. You can lengthen or shorten any side of the border by dragging it with the mouse.

The window corner can be used to drag any two sides of the window.

The mouse pointer is displayed if you have a mouse installed. The mouse is usually in the form of an arrow, an I, or a crosshair (+). For more information, see the section Using the Mouse on page 65.

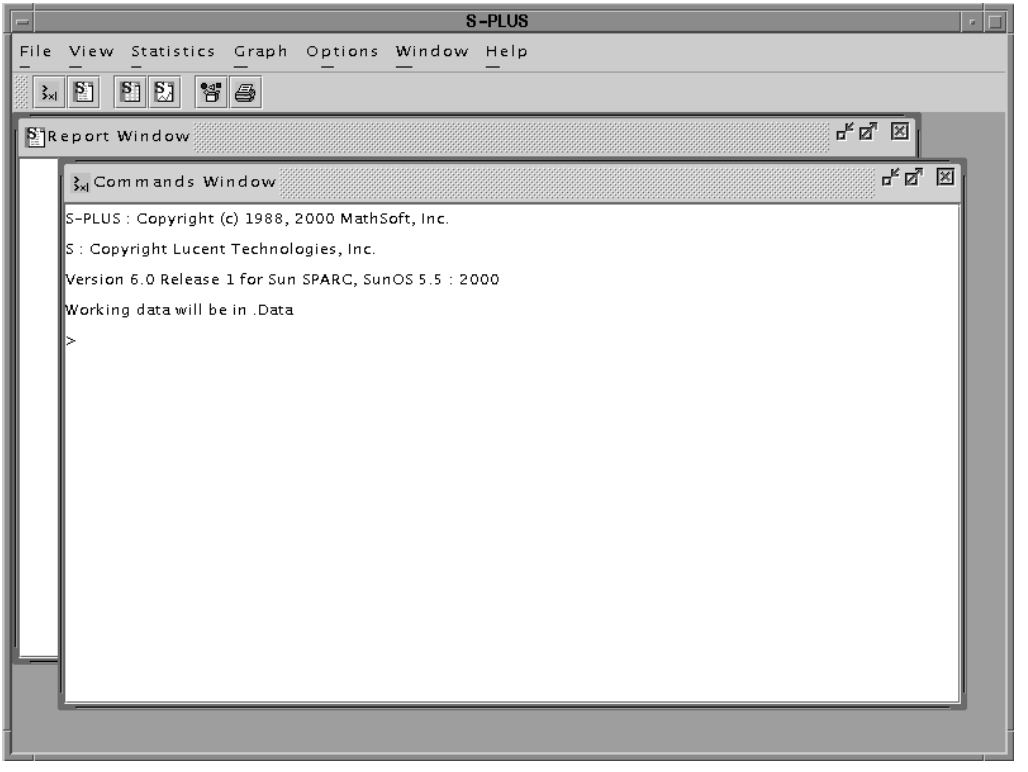

S-PLUS
File View Statistics Graph Options Window Help
Report Window
Commands Window
S-PLUS : Copyright (c) 1988, 2000 MathSoft, Inc.
S : Copyright Lucent Technologies, Inc.
Version 6.0 Release 1 for Sun SPARC, SunOS 5.5 : 2000
Working data will be in .Data
>

Figure 3.2: *The opening main window of S-PLUS includes a* ***Commands*** *window. Notice that the main window has a* ***Control-menu*** *and* ***Minimize*** *and* ***Maximize*** *buttons, while the contained window has* ***Minimize***, ***Maximize*** *and* ***Close*** *buttons (top right). Subwindows can be sized and moved, but only within the confines of the main S-PLUS window.*

Switching to a Different Window

At any time you can have many windows open simultaneously in S-PLUS. The number of windows is limited only by your system's memory resources. To switch from one window to another window, click on any portion of the preferred window that is visible. Alternatively, you can select the preferred window from the list at the bottom of the **Window** menu.

Moving and Sizing Windows

A maximized window cannot be moved or resized. A smaller window can be moved or resized within the confines of the application window. Note that not all windows can be resized.

To move a window or dialog

1. Click in the window or dialog to make it active.
2. Click and drag the title bar until the window or dialog is in the desired location.

To resize a window

1. Click in the window to make it active.
2. Position the mouse over one of the four window borders.
3. The mouse changes to a double-headed arrow when it is over the border.
4. Click and drag the border to the desired size.

To expand a window to maximum size

1. Click in the window to make it active.
2. Click the **Maximize** button on the title bar, or double-click the title bar. Note that the **Maximize** button changes to the **Restore** button.

Viewing Multiple Windows

In S-PLUS, each type of object, such as a graph or data set, is displayed in a separate window. You can also have multiple windows of the same graph or data set open at the same time. You have several options for viewing multiple windows.

To view the windows tiled

- From the **Window** menu, choose **Tile**.

To view the windows layered, with only the title bars visible

- From the **Window** menu, choose **Cascade**.

Closing Windows

To close a window

- Click the **Close** button on the title bar of the window.

To close all open windows

- Double-click the **Control-menu** box, or choose **Exit** from the **File** menu. This closes all open windows and quits S-PLUS.

Using Main Menus

When you choose one of the main menu options, a list of additional options drops down. You can choose any of the options in the list. Menu options with a ► symbol at the end of the line display a submenu when selected. Menu commands with an ellipsis (...) after the command display a dialog box when selected.

To choose a menu option

Point to the desired menu option and click the left mouse button

or

Press the ALT key to access the menu bar, and then press the underlined key in the desired menu option.

To cancel a menu, click outside the menu or press ESC.

Specifying Options in Dialogs

Choosing a menu option often displays a dialog. You can use dialogs to specify information about a particular action. In S-PLUS there are two types of dialogs: action dialogs and property dialogs. Action dialogs carry out commands such as creating a graph. Property dialogs display and allow you to modify the properties and characteristics in your S-PLUS session.

Dialogs can contain multiple, tabbed pages of options. To see the options on a different page of the dialog, click the page name. When you choose **OK** or **Apply** (or press CTRL+ENTER), any changes made on any of the tabbed pages are applied to the selected object.

Most of S-PLUS's dialogs are modeless. They can be moved around on the screen and they remain open until you choose to close them. This means you can make changes in a dialog and see the effect without closing the dialog. This is useful when you are experimenting with

changes to an object and want to see the effect of each change. The **Apply** button can be used to apply changes without closing the dialog. When you are ready to close the dialog, you can either choose **Cancel** or click the **Close** box on the dialog.

Note

Choosing OK closes the dialog and executes the command specified by it. If you do not wish the command to execute after the dialog closes, perhaps because you have already clicked on Apply, choose Cancel instead of OK.

The OK, Cancel, and Apply Buttons

When you are finished setting options in a dialog box, you can click on the **OK**, **Cancel** or **Apply** buttons.

- **OK**: choose the **OK** button or press CTRL+ENTER to close the dialog box and carry out the action.
- **Cancel**: choose the **Cancel** button to close the dialog box and discard any of the changes you have made in the dialog. Sometimes changes cannot be canceled (for example, when changes have made with **Apply**, or when changes have been made outside of the dialog with the mouse).
- **Apply**: choose the **Apply** button to carry out the action without closing the dialog. Most of the S-PLUS dialogs have an **Apply** button, which acts much like an **OK** button except it does not close the dialog box. You can specify changes in the dialog box and then choose the **Apply** button to see your changes, keeping the dialog open so that you can make more changes without having to re-select the dialog.

Typing and Editing in Dialog Boxes

Table 3.2 lists special keys for navigating through and performing tasks in dialog boxes. In addition, many dialogs contain text edit boxes, which allow you to type in information such as file names and graph titles.

Table 3.2: *Shortcut keys in dialog boxes.*

Action	Special Keys
Move to the next option in the dialog	TAB
Move to a specific option and select it	ALT+underlined letter in the option name. Press again to move to additional options with the same underlined letter.
Display a drop-down list	DOWN direction key
Select an item from a list	UP or DOWN direction keys to move, ENTER key to close the list

To replace text in a dialog

1. Select the existing text with the mouse, or press ALT+underlined letter in the option name.
2. Type the new text. Any highlighted text is immediately overwritten when you begin typing the new text.

To edit text in a text box

1. Position the insertion point in the text box. If text is highlighted, it will be replaced when you begin typing.
2. Edit the text.

Using Toolbar Buttons

Toolbars contain buttons that are shortcuts to menu selections. You can use toolbar buttons to perform file operations such as opening a new **Graph** window or printing a window. To select a toolbar button, position the mouse pointer over the desired button and click. For example, you can print your current **Graph** window by clicking on the **Print** button.

S-PLUS WINDOWS

The S-PLUS user interface contains five types of windows: the **Objects Summary**, **Data Viewer**, **Graph** window, **Commands** window, and **Report** window. These windows allow you to easily organize your work session, work with data and graphs simultaneously, and automate repetitive tasks.

Objects Summary

The **Objects Summary** window, shown in Figure 3.3, gives a brief overview of the objects in your working database. To open an **Objects Summary** window in your S-PLUS session, select **View ► Objects Summary**.

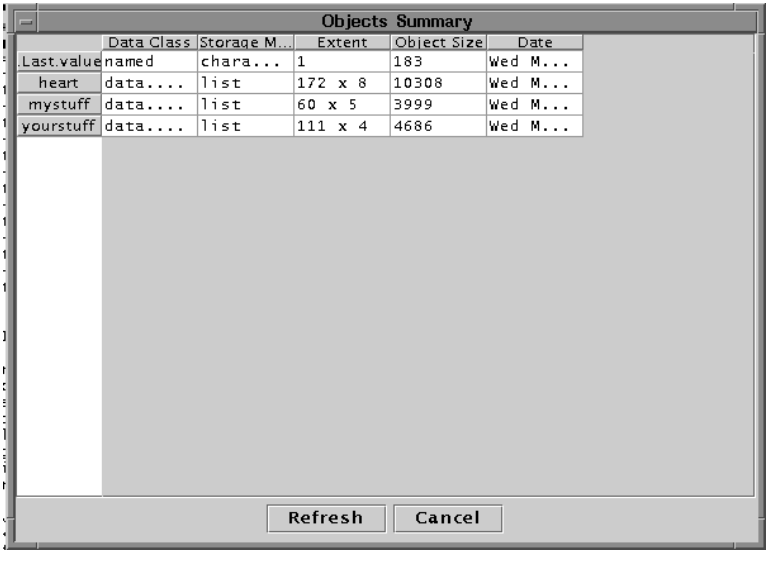


Figure 3.3: *An **Objects Summary** window: several can be open simultaneously.*

Data Viewer

The **Data Viewer,** shown in Figure 3.4, displays data sets in a non-editable tabular format. To view a data set, select **View ► New Data Viewer** from the main menu. A dialog appears that prompts you for the name of an S-PLUS data set. If the data set is in your working database, you can select its name from the pull-down list; otherwise, type the name directly in the **Data Set** field and click **OK**.

.

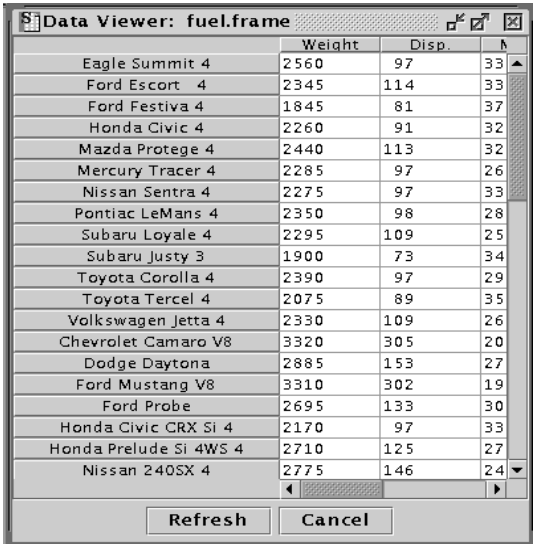

Data Viewer: fuel.frame
Weight
Disp.
Eagle Summit 4 2560 97 33
Ford Escort 4 2345 114 33
Ford Festiva 4 1845 81 37
Honda Civic 4 2260 91 32
Mazda Protege 4 2440 113 32
Mercury Tracer 4 2285 97 26
Nissan Sentra 4 2275 97 33
Pontiac LeMans 4 2350 98 28
Subaru Loyale 4 2295 109 25
Subaru Justy 3 1900 73 34
Toyota Corolla 4 2390 97 29
Toyota Tercel 4 2075 89 35
Volkswagen Jetta 4 2330 109 26
Chevrolet Camaro V8 3320 305 20
Dodge Daytona 2885 153 27
Ford Mustang V8 3310 302 19
Ford Probe 2695 133 30
Honda Civic CRX Si 4 2170 97 33
Honda Prelude Si 4WS 4 2710 125 27
Nissan 240SX 4 2775 146 24
Refresh
Cancel

Figure 3.4: *The **Data Viewer**.*

It is important to note that only objects of class `"data.frame"` are recognized by the dialogs in the S-PLUS graphical user interface. This means that the **Data Viewer** cannot find or display matrices, vectors, or time series objects; to display objects of these types, you must first convert them to class `"data.frame"`.

Graph Window

By default, S-PLUS displays graphics in a Java graphics window, as shown in Figure 3.5. Each **Graph** window can contain one or more graphs, and you can work with multiple graph windows in your S-PLUS session.

There are four different ways to create a graphics window:

1. Generate plots from the dialogs in the **Graph** menu.
2. Generate plots from functions called in the **Commands** window.
3. Select **View ► New Graph Window**, or click on the **New Graph Window** toolbar button. This opens a blank graphics window.
4. Explicitly call the `java.graph()` device in the **Commands** window, which also opens a blank graphics window.

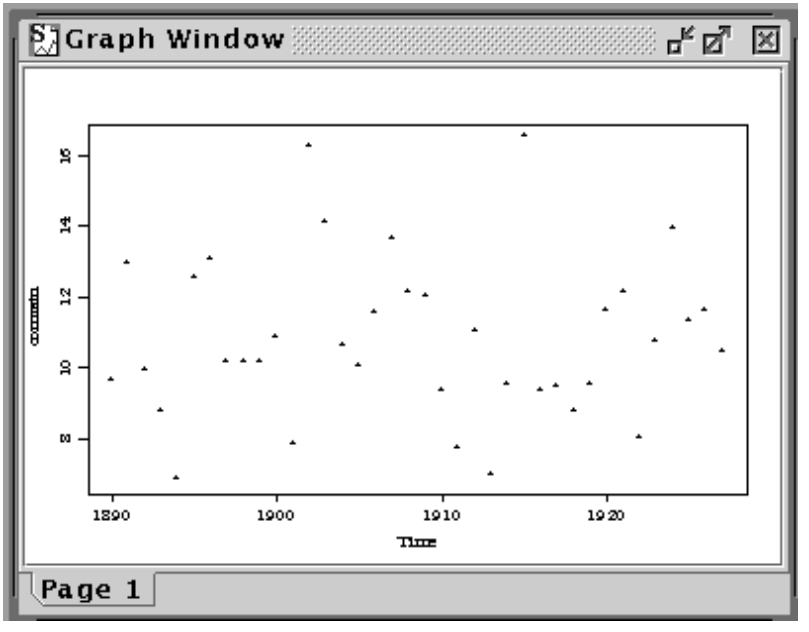


Figure 3.5: *A **Graph** window displaying a Trellis graph.*

Commands Window

The **Commands** window allows you to access the powerful S-PLUS programming language. You can modify existing functions or create new ones tailored to your specific analysis needs by using the **Commands** window. By default, the **Commands** window is open when you start S-PLUS. See the chapter Getting Started for examples of typing expressions and working from the **Commands** window.

Report Window

When a dialog is launched, output is directed to the **Report** window, shown in Figure 3.6. Text in the **Report** window can be formatted before cutting and pasting it into another application. The **Report** window is a place-holder for the text output resulting from any operation in S-PLUS. For example, error messages and warnings are sometimes placed in a **Report** window.

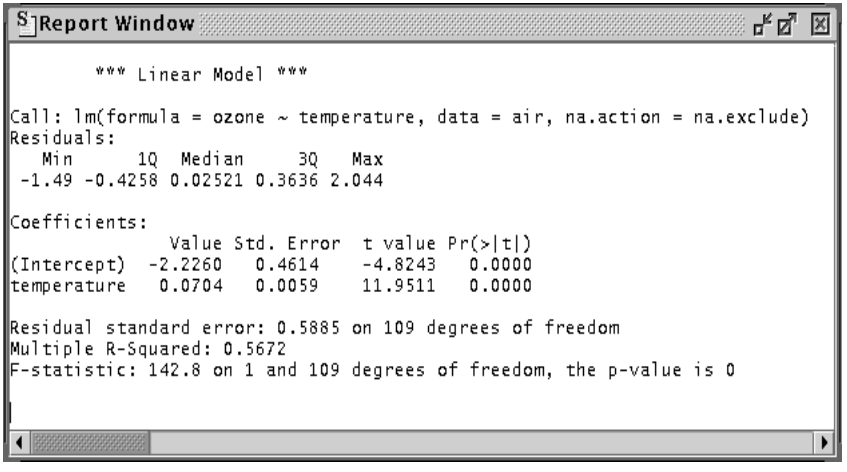

Report Window
*** Linear Model ***
Call: lm(formula = ozone ~ temperature, data = air, na.action = na.exclude)
Residuals:
Min 1Q Median 3Q Max
-1.49 -0.4258 0.02521 0.3636 2.044
Coefficients:
Value Std. Error t value Pr(>|t|)
(Intercept) -2.2260 0.4614 -4.8243 0.0000
temperature 0.0704 0.0059 11.9511 0.0000
Residual standard error: 0.5885 on 109 degrees of freedom
Multiple R-Squared: 0.5672
F-statistic: 142.8 on 1 and 109 degrees of freedom, the p-value is 0

Figure 3.6: *A **Report** window is an option for holding textual output.*

S-PLUS Menus

When you choose one of the main menu options, a list of additional options drops down. You can choose from any of the active options in the list. Menu options with a ▶ symbol at the end of the line display submenus when selected. Menu items with an ellipsis (...) after the command display a dialog when selected. Table 3.3 gives brief descriptions of each of the main S-PLUS menus.

Table 3.3: *The main S-PLUS menus.*

Main menu	Notes
File	Importing, exporting, saving, and printing files.
View	Standard options such as whether the Commands and Report windows are open.
Statistics	See the Statistics chapter.
Graph	See the Menu Graphics chapter.
Options	General settings for options, styles, and color schemes.
Window	Standard windows controls such as Cascade and Tile.
Help	Gives on-line access to the S-PLUS help system.

S-PLUS Dialogs

Dialogs can contain multiple tabbed pages of options, as shown in Figure 3.7. To see the options on a different page of the dialog, check the page name. When you choose **OK** or **Apply**, any changes made on any of the tabbed pages are applied to the selected objects.

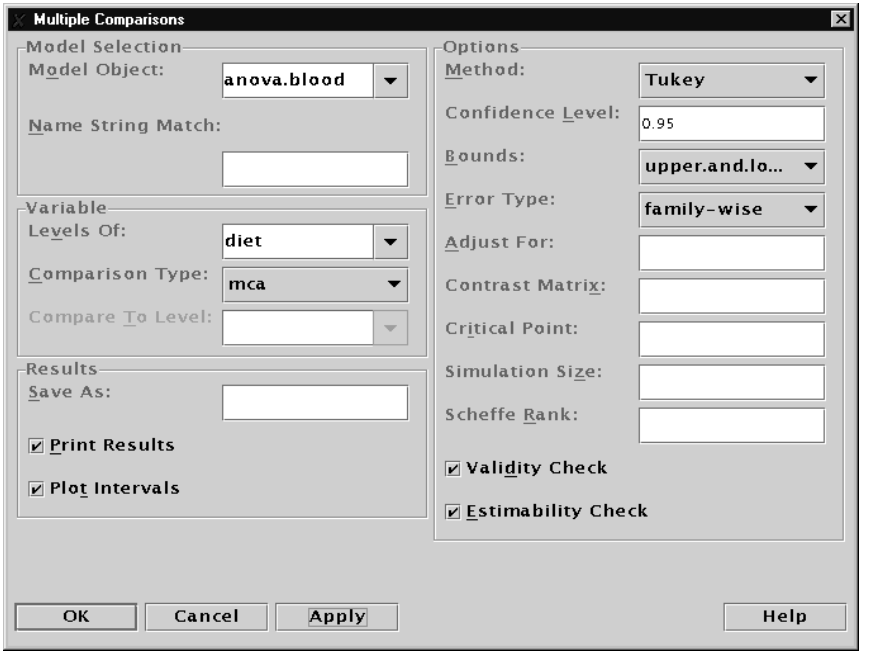


Figure 3.7: *An S-PLUS dialog for performing multiple comparisons.*

IMPORTING AND EXPORTING DATA

4

Introduction **80**

Dialogs **81**
- The **Import Data** Dialog 81
- Filtering Rows 86
- Format Strings 88
- The **Export Data** Dialog 89

Supported File Formats **93**

Examples **95**
- Importing and Exporting Subsets of Data 95
- Importing and Exporting Character Data 98

INTRODUCTION

S-PLUS can read a wide variety of data formats, which makes importing data straightforward. S-PLUS also allows you to export data sets for use in other applications. The primary tools for importing and exporting data are command-line functions named `importData` and `exportData`, respectively. In the graphical user interface, these functions are implemented in the **Import Data** and **Export Data** dialogs. We discuss the dialogs and their options in this chapter; for detailed discussions on the functions themselves, see the online help files or the *Programmer's Guide*.

DIALOGS

The Import Data Dialog

To import data from the graphical user interface, select **File ▶ Import Data**. The **Import Data** dialog appears, as shown in Figure 4.1.

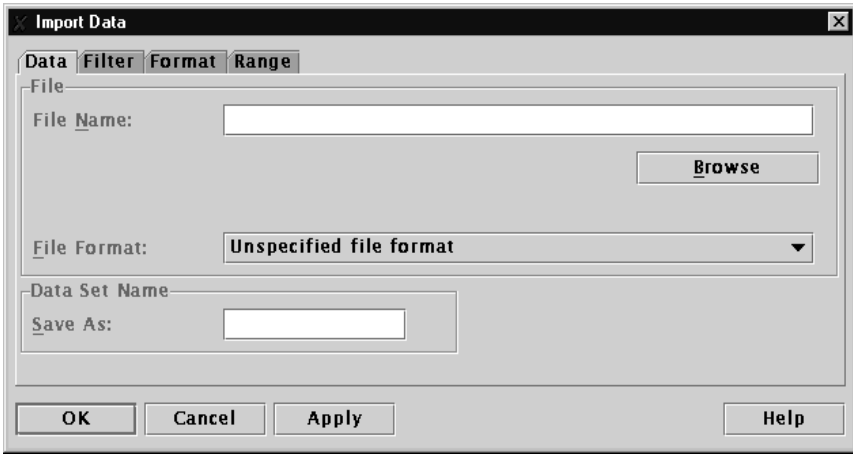


Figure 4.1: *The **Data** page of the **Import Data** dialog.*

The Data page

The **Data** page, shown in Figure 4.1, allows you to navigate to a particular directory, choose the file to be imported, specify a particular file format, and name the S-PLUS object in which the data should be stored. Descriptions of the individual fields are:

- **File Name**: Select or type the name of the file to import. To navigate to the directory that contains your data file, click on the **Browse** button.
- **File Format**: Select the format of the file to import. See the section Supported File Formats for details on the selections in this list.
- **Save As**: Enter a valid name for the S-PLUS object in which the data should be stored. If an object with this name already exists, its contents are overwritten. A valid name is any combination of alphanumeric characters (including the period character ".") that does not start with a number. Names are case-sensitive, so X and x refer to different objects.

Note

By default, the **Import Data** dialog looks for files in your current working directory, which is one level up from your **.Data** directory. If the file you wish to import is located in another directory, either click on the **Browse** button to search for it, or explicitly type the path to the file in the **File Name** field.

The Filter page

The **Filter** page, shown in Figure 4.2, allows you to subset the data to be imported. By specifying a query, or *filter* expression, you gain additional functionality; it is possible to import random samples of your data using a filter, for example. By default, the import filter is blank and thus imports all of the data. Descriptions of the individual fields are given below.

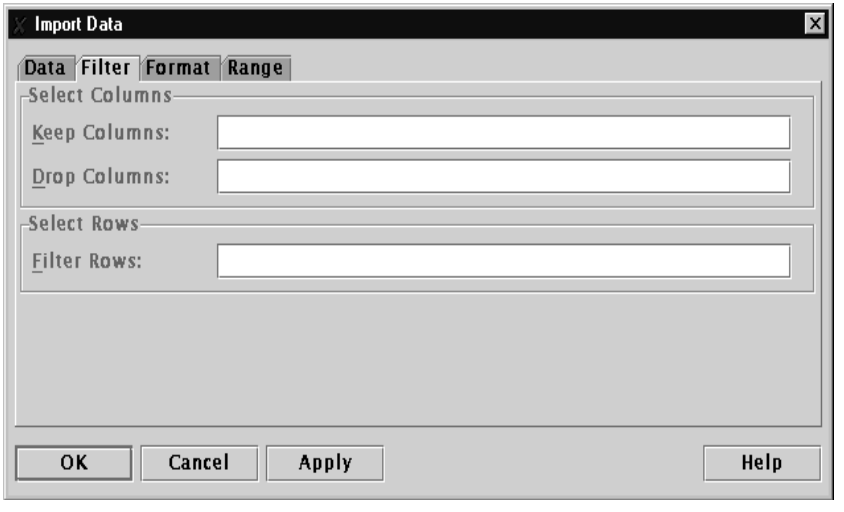


Figure 4.2: *The **Filter** page of the **Import Data** dialog.*

- **Keep Columns**: Specify a character vector of column names or numeric vector of column numbers that should be imported from the data file. Only one of **Keep Columns** and **Drop Columns** can be specified.
- **Drop Columns**: Specify a character vector of column names or numeric vector of column numbers that should *not* be imported from the data file. Only one of **Keep Columns** and **Drop Columns** can be specified.

- **Filter Rows**: Specify a logical expression for selecting the rows that should be imported from the data file. See the section Filtering Rows for a description of the syntax accepted by this field.

The Format page

The **Format** page, shown in Figure 4.3, contains options specific to ASCII, SAS, and SPSS data files. In addition, the **Format** page allows you to specify the data types of imported character expressions. Descriptions of the individual fields are given below.

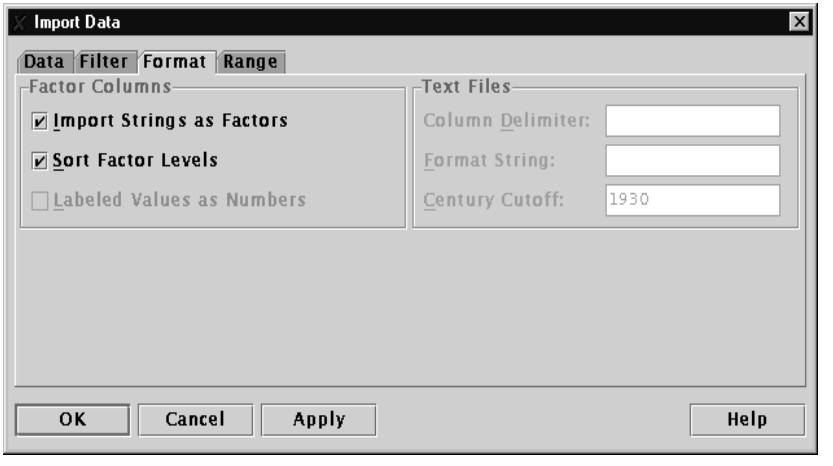


Figure 4.3: *The **Format** page of the **Import Data** dialog.*

- **Import Strings as Factors**: If this option is selected, then all character strings are converted to factor variables when the data file is imported. Otherwise, they are imported with the data class `"character"`.
- **Sort Factor Levels**: If this option is selected, then S-PLUS (alphabetically) sorts the levels for all factor variables that are created from character strings. Otherwise, the levels are defined in the order they are read in from the data file.
- **Labeled Values as Numbers**: If this option is selected, then SAS and SPSS variables that have labels are imported as numbers. Otherwise, the value labels are imported.
- **Column Delimiter**: When importing an ASCII text file, this field specifies the character delimiters to use. The expressions `\n` and `\t` are the only multi-character delimiters allowed, and denote a newline and a tab, respectively. Double quotes

are reserved characters, and therefore cannot be used as standard delimiters. If a delimiter is not supplied, S-PLUS searches the file automatically for the following, in the order given: tabs, commas, semicolons, and vertical bars. If none of these are detected, blank spaces are treated as delimiters.

- **Format String**: This field is required when importing a formatted ASCII text file (FASCII). A *format string* specifies the data types and formats of the imported columns. For more details on the syntax accepted by this field, see the section Format Strings.
- **Century Cutoff**: When importing an ASCII text file, this field specifies the origin for two-digit dates. Dates with two-digit years are assigned to the 100-year span that starts with this numeric value. The default value of 1930 thus reads the date `6/15/30` as June 15, 1930, while the date `12/29/29` is interpreted as December 29, 2029.

The Range page

The **Range** page, shown in Figure 4.4, contains options that allow you to filter rows and columns when importing data from a spreadsheet (Excel and Lotus files, etc.). Descriptions of the individual fields are given below.

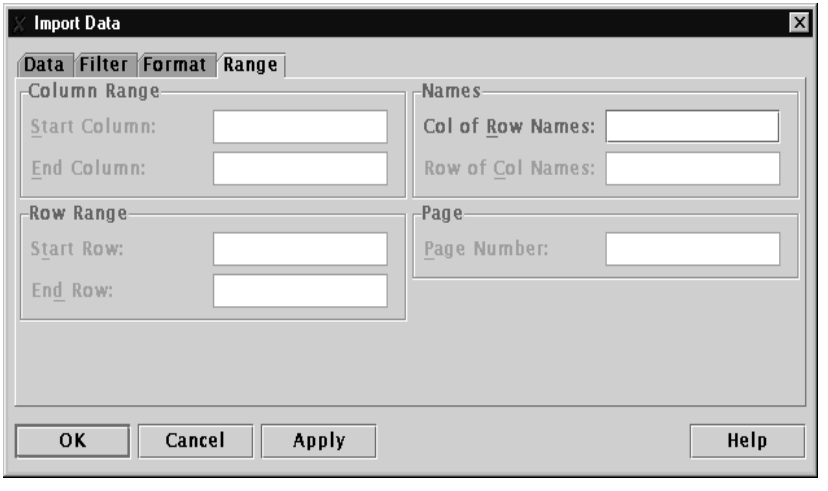


Figure 4.4: *The **Range** page of the **Import Data** dialog.*

- **Start Column**: Specify an integer that corresponds to the first column to be imported from the spreadsheet. For example, a value of 5 causes S-PLUS to begin reading data from the file at column 5. By default, the first column in the spreadsheet is used.
- **End Column**: Specify an integer that corresponds to the final column to be imported from the spreadsheet. By default, the final column in the spreadsheet is used, and S-PLUS imports everything that follows the **Start Column**.
- **Start Row**: Specify an integer that corresponds to the first row to be imported from the spreadsheet. For example, a value of 10 causes S-PLUS to begin reading data from the file at row 10. By default, the first row in the spreadsheet is used.
- **End Row**: Specify an integer that corresponds to the final row to be imported from the spreadsheet. By default, the final row in the spreadsheet is used, and S-PLUS imports everything that follows the **Start Row**.
- **Col of Row Names**: Specify an integer denoting the column of the data file that should be used for row names. The chosen column is not included in the S-PLUS data set that gets created. You can use this option with ASCII text files as well as with spreadsheets.
- **Row of Col Names**: Specify an integer denoting the row of the data file that should be used for column names. The chosen row is not included in the S-PLUS data set that gets created. By default, S-PLUS attempts to formulate sensible column names from the first imported row.
- **Page Number**: Specify the page number of the spreadsheet that should be imported.

Note

Because the underscore "_" is a reserved character in S-PLUS, the **Import Data** dialog converts all column names that have underscores in them so that they contain periods "." instead.

Filtering Rows

The **Filter Rows** field in the **Import Data** dialog accepts logical expressions that specify the rows to be imported from the data file. The filter must be written in terms of the original column names in the file, and not in terms of the variable names specified by the **Row of Col Names** field. Note that the filter is *not* evaluated by S-PLUS. This means that expressions containing built-in S-PLUS functions (such as `mean`) are not allowed. One special exception to this rule deals with missing values: you can use `NA` to denote missing values in the logical expressions, though you cannot use `NA`-specific functions such as `is.na` and `na.exclude`.

Table 4.1 lists the logical operators that are accepted by the **Filter Rows** field. Thus, to select all rows that do not have missing values in the `id` column, type `id != NA`. To import all rows corresponding to 10-year-old children who weigh less than 150 pounds, type `Age==10 & Weight<150`. In the filter expression, the variable name should be on the left side of the logical operator; i.e., type `Age > 12` instead of `12 < Age`.

Table 4.1: *Logical operators accepted by the **Filter Rows** field.*

Operator	Description
==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
&	logical and
|	logical or
!	negation

The wildcard characters `?` (for single characters) and `*` (for strings of arbitrary length) can be used to select subgroups of character variables. For example, the logical expression `account == ?????22` selects all rows for which the `account` variable is six characters long and ends in 22. The expression `id == 3*` selects all rows for which `id` starts with 3, regardless of the length of the string.

You can use the built-in variable `@rownum` to import specific row numbers. For example, the expression `@rownum < 200` imports the first 199 rows of the data file.

Sampling functions

Three functions that permit random sampling of your data are available to use in a **Filter Rows** expression:

- `samp.rand`: accepts a single numeric argument `prop`, where $0 \leq$ `prop` ≤ 1. Rows are selected randomly from the data file with a probability of `prop`.
- `samp.fixed`: accepts two numeric arguments, `sample.size` and `total.observations`. The first row is drawn from the data file with a probability of `sample.size`/`total.observations`. The *i*th row is drawn with a probability of (`sample.size` - *i*)/(`total.observations` - *i*), where $i = 1, 2, \ldots,$ `sample.size`.
- `samp.syst`: accepts a single numeric argument `n`. Every *n*th row is selected systematically from the data file after a random start.

Expressions are evaluated from left to right, so you can sample a subset of the rows in your data file by first subsetting, and then sampling. For example, to import a random sample of half the rows corresponding to high school graduates, use the expression `schooling>=12 & samp.rand(0.5)`.

The sampling functions use the S-PLUS random number generator to create random samples. You can therefore use the `set.seed` function in the **Commands** window to produce the same data sample repeatedly. For more details, see the help files for `set.seed` and `.Random.seed`.

Format Strings

Format strings are used when importing data from, or exporting data to, fixed-format text files (FASCII). With a format string, you specify how each character in the imported file should be treated. You must use a format string, together with the FASCII file type, if the columns in your data file are not separated by delimiters.

The Import Data dialog

In the **Import Data** dialog, a valid format string includes a percent `%` sign followed by the data type, for each column in the data file. Available data types are: `s`, which denotes a character string; `f`, which denotes a numeric value; and the asterisk `*`, which denotes a skipped column. One of the characters specified in the **Column Delimiters** field must separate each specification in the string. For example, the format string

```
%s, %f, %*, %f
```

imports the first column of the data file as type `"character"`, the second and fourth columns as `"numeric"`, and skips the third column altogether.

If a variable is designated as `"numeric"` and the value of a cell cannot be interpreted as a number, the cell is filled in with a missing value. Incomplete rows are also filled in with missing values.

Note

Some dates in text files may be imported automatically as numbers. After importing data that contain dates, you should check the class of each column in S-PLUS, and change them to the appropriate data types if necessary.

Note that format strings and field width specifications are irrelevant for regular ASCII files, and are therefore ignored. For fixed-format ASCII text files, however, you can specify an integer that defines the width of each field. For example, the format string

```
%4f, %6s, %3*, %6f
```

imports the first four entries in each row as a numeric column. The next six entries in each row are read as characters, the next three are skipped, and then six more entries are imported as another character column.

The Export Data dialog

When exporting to a fixed-format ASCII text file, the syntax accepted by the **Format String** field is similar to the **Import Data** option. In addition to the data type, however, the precision of numeric values can also be specified. For example, the format string

```
%3, %7.2, %4, %5.2
```

exports the first and third columns as whole numbers with 3 and 4 digits, respectively. The second and fourth columns each have two decimal digits of precision. The precision value is ignored if it is given for a character column; if the precision is not specified, is assumed to be zero. If you export row names for your data set, the first entry in the format string is reserved for the row names.

Specifying a format string can potentially speed up the export of data sets that have many character columns. If a format string is not specified, S-PLUS must check the width of every entry in a character or factor column, and determine a width large enough for all values in the column. Since many of the supported file types use fixed widths, considerable space can be saved by specifying a narrow width for character columns that have many short values and only a few long values; with this approach, the few long values are truncated.

The Export Data Dialog

To export data from the graphical user interface, select **File ▶ Export Data**. The **Export Data** dialog appears, as shown in Figure 4.5.

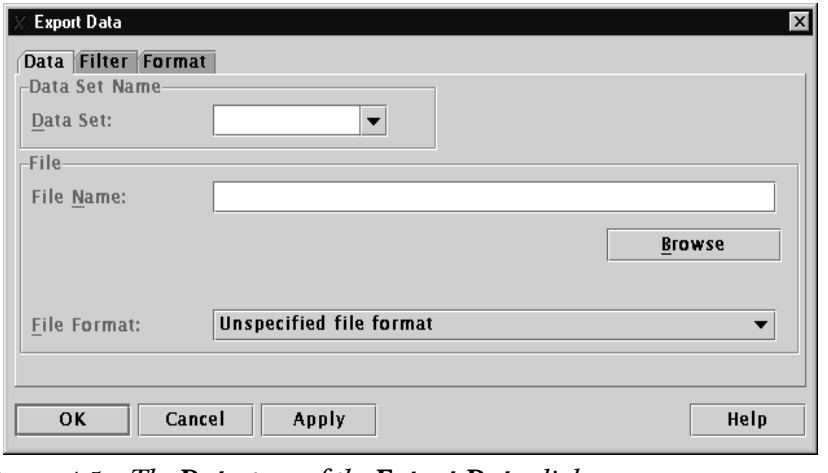


Figure 4.5: *The **Data** page of the **Export Data** dialog.*

The Data page

The **Data** page, shown in Figure 4.5, allows you to name the S-PLUS object to be exported, navigate to the directory in which the file should be stored, and specify a particular file format. Descriptions of the individual fields are given below.

- **Data Set**: Enter the name of the S-PLUS object to be exported. Names are case-sensitive, so X and x refer to different objects.
- **File Name**: Select or type the name of the file that should contain the contents of the data set. S-PLUS notifies you if the file already exists, and then gives you the opportunity to either overwrite the file's contents or cancel the export. To navigate to a particular directory, click on the **Browse** button.
- **File Format**: Select the format of the exported data file. See the section Supported File Formats for details on the selections in this list..

Note

By default, the **Export Data** dialog saves files in your current working directory, which is one level up from your **.Data** directory. If you wish to export a file to another directory, either click on the **Browse** button to search for it, or explicitly type the path to the file in the **File Name** field.

The Filter page

The **Filter** page, shown in Figure 4.6, allows you to subset the data to be exported. By specifying a filter expression, you gain additional functionality; it is possible to export random samples of your data using a filter, for example. By default, the export filter is blank and thus exports all of the data. Descriptions of the individual fields are given below.

- **Keep Columns**: Specify a character vector of column names or numeric vector of column numbers that should be exported from the data set. Only one of **Keep Columns** and **Drop Columns** can be specified.
- **Drop Columns**: Specify a character vector of column names or numeric vector of column numbers that should *not* be exported from the data set. Only one of **Keep Columns** and **Drop Columns** can be specified.

- **Filter Rows**: Specify a logical expression for selecting the rows that should be exported from the data set. See the section Filtering Rows for a description of the syntax accepted by this field. Although the discussion in that section is specific to the **Import Data** dialog, the descriptions are analogous for the **Export Data** dialog.

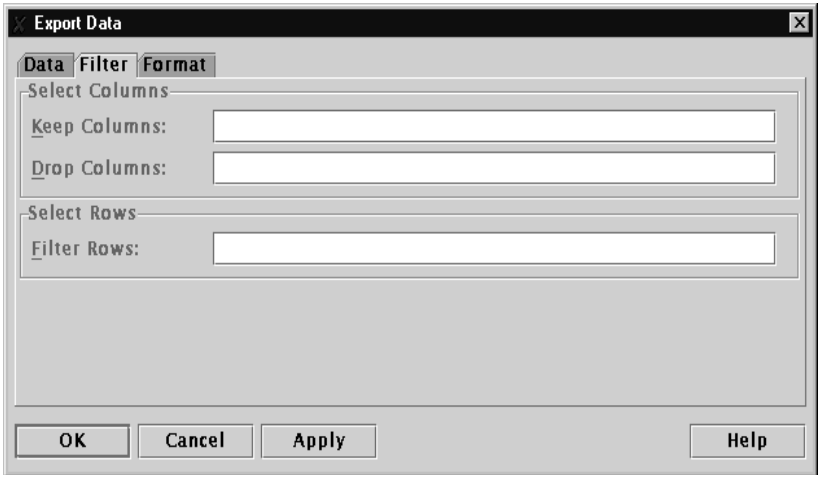


Figure 4.6: *The **Filter** page of the **Export Data** dialog.*

The Format page

The **Format** page, shown in Figure 4.7, contains options specific to ASCII text files and factor variables. In addition, the **Format** page allows you to specify whether row names and column names should be exported from your data set. Descriptions of the individual fields are given below.

- **Export Column Names:** If this option is selected, then S-PLUS includes the column names of the data set as the first row in the file.
- **Export Row Names**: If this option is selected, then S-PLUS includes the row names of the data set as the first column in the file.
- **Quote Character Strings**: If this option is selected, then all factors and character variables in the data set are exported with quotation marks, so that they are recognized as strings.

- **Column Delimiter**: When exporting to an ASCII text file, this field specifies the character delimiters to use. The expressions `\n` and `\t` are the only multi-character delimiters allowed, and denote a newline and a tab, respectively. Double quotes are reserved characters, and therefore cannot be used as standard delimiters. By default, S-PLUS uses commas as delimiters.
- **Format String**: When exporting to an ASCII text file, this field specifies the data types and formats of the exported columns. For more details on the syntax accepted by this field, see the section Format Strings.

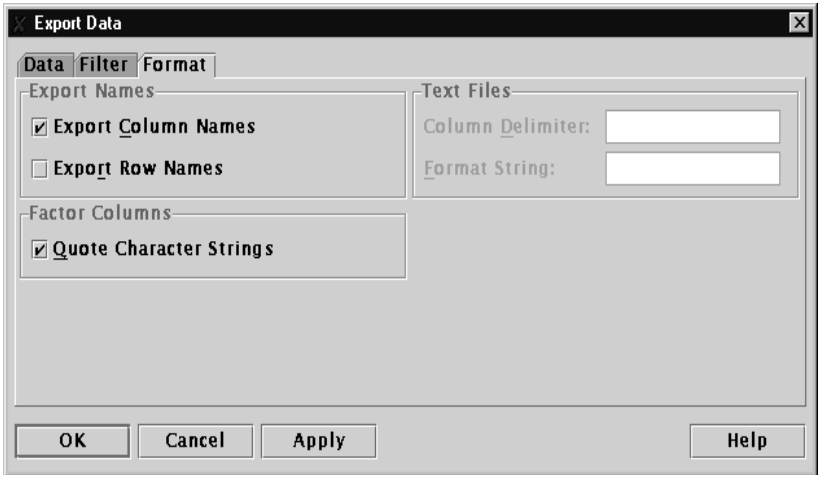


Figure 4.7: *The **Format** page of the **Export Data** dialog.*

SUPPORTED FILE FORMATS

Table 4.2 lists the file types supported by the **Import Data** dialog. In addition to all the listed formats, S-PLUS also exports to HTML tables, which have a default suffix of ".htm".

Table 4.2: *Supported file formats for the **Import Data** and **Export Data** dialogs.*

File Type	Standard Suffix	Notes
ASCII Text File	comma delimited: .csv all other delimiters: .asc, .dat, .txt, .prn	
dBase	.dbf	
Microsoft Excel Worksheet	.xls	Must be Excel version 4 or earlier.
Formatted ASCII (FASCII) Text File	.fix, .fsc	
Gauss/Gauss96 File	.dat	
Lotus 1-2-3 Worksheet	.wks, .wk1, .wk3, .wk4, .wrk	
Matlab Matrix	.mat	Matlab version 5 files are accepted, along with earlier file formats. The Matlab file can contain only one matrix.
Minitab Workbook	.mtw	
Quattro Pro Worksheet	.wq1, .wb2, .wb3	
SAS	version 7 or 8: .sas7bdat, .sd7 Windows/OS2: .sd2 HP, IBM, & Sun Unix: .ssd01 DEC Unix: .ssd04 Transport File: .tpt, .xpt	Variable names are converted to lower-case letters when imported.

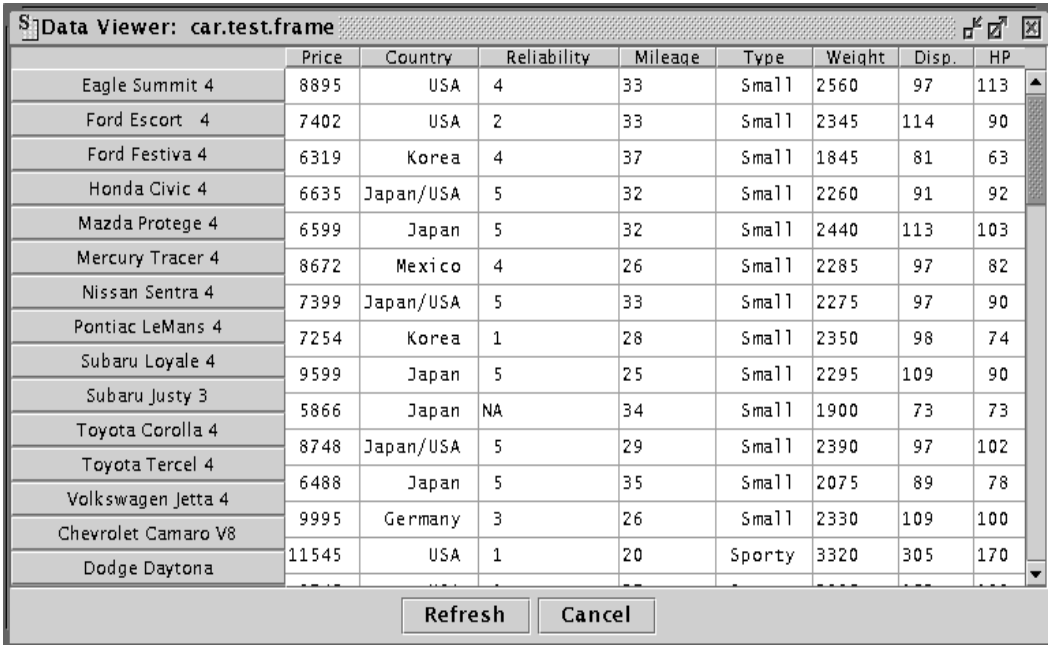
Table 4.2: *Supported file formats for the* ***Import Data*** *and* ***Export Data*** *dialogs.*

File Type	Standard Suffix	Notes
SPSS	Regular Data File: .sav Portable Data File: .por	Variable names are converted to lower-case letters when imported.
Stata Data File	.dta	
Systat File	.syd, .sys	

EXAMPLES

Importing and Exporting Subsets of Data

In the following examples, we import and export subsets of the built-in data set `car.test.frame`, using the options in the **Filter** page of the **Import Data** and **Export Data** dialogs. The `car.test.frame` data is taken from the April 1990 issue of *Consumer Reports*, and contains 60 observations (rows) and 8 variables (columns). Observations of price, manufacturing country, reliability, mileage, type, weight, engine displacement, and horsepower were taken for each of sixty cars. This data set is shown in Figure 4.8.



	Price	Country	Reliability	Mileage	Type	Weight	Disp.	HP
Eagle Summit 4	8895	USA	4	33	Small	2560	97	113
Ford Escort 4	7402	USA	2	33	Small	2345	114	90
Ford Festiva 4	6319	Korea	4	37	Small	1845	81	63
Honda Civic 4	6635	Japan/USA	5	32	Small	2260	91	92
Mazda Protege 4	6599	Japan	5	32	Small	2440	113	103
Mercury Tracer 4	8672	Mexico	4	26	Small	2285	97	82
Nissan Sentra 4	7399	Japan/USA	5	33	Small	2275	97	90
Pontiac LeMans 4	7254	Korea	1	28	Small	2350	98	74
Subaru Loyale 4	9599	Japan	5	25	Small	2295	109	90
Subaru Justy 3	5866	Japan	NA	34	Small	1900	73	73
Toyota Corolla 4	8748	Japan/USA	5	29	Small	2390	97	102
Toyota Tercel 4	6488	Japan	5	35	Small	2075	89	78
Volkswagen Jetta 4	9995	Germany	3	26	Small	2330	109	100
Chevrolet Camaro V8	11545	USA	1	20	Sporty	3320	305	170
Dodge Daytona								

Figure 4.8: *The `car.test.frame` data in a* ***Data Viewer****.*

Using the Keep Columns and Drop Columns options

1. Open the **Export Data** dialog.
2. Type `car.test.frame` in the **Data Set** field. Type **car.keep.txt** in the **File Name** field, and choose **ASCII file – tab delimited** from the **File Format** list.
3. Click on the **Filter** tab and type **2,3,5** in the **Keep Columns** field.
4. Click on the **Format** tab and check the **Export Row Names** box.
5. Click **OK**.

S-PLUS creates a tab-delimited text file named **car.keep.txt** in your working directory. The file contains the row names in `car.test.frame` in addition to the three specified columns: `Price`, `Country`, and `Mileage`. Because we checked the **Export Row Names** box, the row names are considered the first column in the exported data set. This is why a **Keep Columns** value of **2,3,5** actually exports the first, second, and fourth variables in the data set.

The syntax for the **Drop Columns** field is similar, as the following example shows.

1. Open the **Import Data** dialog.
2. Type **car.keep.txt** in the **File Name** field, and choose **ASCII file – tab delimited** from the **File Format** list. Type `car.drop` in the **Save As** field.
3. Click on the **Filter** tab and type **Country, Mileage** in the **Drop Columns** field. This imports all columns from the text file except those named `Country` and `Mileage`.
4. Click on the **Range** tab and type **1** in the **Col of Row Names** field. This forces S-PLUS to use the first column in the text file as the row names in the data frame.
5. Click **OK**.

The `car.drop` data set, shown in Figure 4.9, contains only the pricing data from `car.test.frame`. Whether used in the **Import Data** or **Export Data** dialog, the **Keep Columns** and **Drop Columns** fields can be specified as either a list of column numbers or a list of variable names.

Figure 4.9: *The* `car.drop` *data set in a* ***Data Viewer****.*

Using the Filter Rows option

1. Open the **Export Data** dialog.
2. Type `car.test.frame` in the **Data Set** field. Type **car.filter.xls** in the **File Name** field, and choose **Excel Worksheet** from the **File Format** list.
3. Click on the **Filter** tab and type **Price < 10000 & Mileage > 27** in the **Filter Rows** field.
4. Click on the **Format** tab and check the **Export Row Names** box.
5. Click **OK**.

S-PLUS creates an Excel file named **car.filter.xls** in your working directory. The file contains the 11 observations from `car.test.frame` for which the `Price` variable is less than $10,000 and the `Mileage` variable is greater than 27 miles per gallon.

Importing and Exporting Character Data

To illustrate the options relating to character data in the **Import Data** and **Export Data** dialogs, we create a simple data set named `animal`. The following S-PLUS command generates a data frame that has five entries: `dog`, `cat`, `bird`, `hyena`, and `goat`:

```
> animal <- data.frame(c("dog", "cat", "bird", "hyena",
+ "goat"))
> animal

      X1
1    dog
2    cat
3   bird
4  hyena
5   goat
```

We can export the text file with the following steps:

1. Open the **Export Data** dialog.
2. Type `animal` in the **Data Set** field and **animal.txt** in the **File Name** field. Select **ASCII file – space delimited** from the **File Format** list.
3. Click on the **Format** tab and deselect the **Export Column Names** option.
4. Click **OK**.

S-PLUS creates a text file named **animal.txt** in your working directory that contains the five entries from `animal`, each with a set of surrounding quotes. The following steps import the data into S-PLUS as character strings:

1. Open the **Import Data** dialog.
2. Type **animal.txt** in the **File Name** field, and select **ASCII file – space delimited** from the **File Format** list. Type `animal.char` in the **Data Set** field.
3. Click on the **Format** tab, and deselect the **Import Strings as Factors** and **Sort Factor Levels** options.
4. Click **Apply**.

S-PLUS recognizes `animal.char` as having data class `"AsIs"`:

```
> animal.char

   Col1
1   dog
2   cat
3  bird
4 hyena
5  goat

> data.class(animal.char$Col1)

[1] "AsIs"
```

To formally convert the `animal.char` column, we can use the `character` or `as.character` functions.

The steps below import the **animal.txt** data as a factor variable:

1. Click on the **Data** tab in the open **Import Data** dialog, and type `animal.fac` in the **Data Set** field.
2. Click on the **Format** tab. Select the **Import Strings as Factors** option, but leave **Sort Factor Levels** box unchecked.
3. Click **Apply**.

The `animal.fac` object is identical to `animal.char`, but S-PLUS now interprets the data as a factor variable:

```
> data.class(animal.fac$Col1)

[1] "factor"

> levels(animal.fac$Col1)

[1] "dog"   "cat"   "bird"  "hyena" "goat"
```

Note that the levels of the factor appear in the same order as they do in the text file. The steps given below sort the levels alphabetically instead.

1. Click on the **Data** tab in the open **Import Data** dialog, and type `animal.fac2` in the **Data Set** field.
2. Click on the **Format** tab and select the **Sort Factor Levels** option.
3. Click **OK**.

The levels of the factor variable are now sorted alphabetically:

```
> data.class(animal.fac2$Col1)

[1] "factor"

> levels(animal.fac2$Col1)

[1] "bird"  "cat"   "dog"   "goat"  "hyena"
```

DATA FRAMES 5

Introduction **102**

The Benefits of Data Frames **103**

Creating Data Frames **104**

Combining Data Frames **109**
Combining Data Frames by Column 110
Combining Data Frames by Row 112
Merging Data Frames 113

Applying Functions to Subsets of a Data Frame **116**
Summaries for Variables by Subsets of Rows 116

Adding New Classes of Variables to Data Frames **123**

INTRODUCTION

Data frames are data objects designed primarily for data analysis and modeling. You can think of them as *generalized* matrices–generalized in a way different from the way arrays generalize matrices. Arrays generalize the *dimensional* aspect of a matrix; data frames generalize the *mode* aspect of a matrix. Matrices can be of only one mode (for example, `"logical"`, `"numeric"`, `"complex"`, `"character"`). Data frames, however, allow you to mix modes from column to column. For example, you could have a column of `"character"` values, a column of `"numeric"` values, a column of categorical values, and a column of `"logical"` values. Each column of a data frame corresponds to a particular variable; each row corresponds to a single "case" or set of observations.

THE BENEFITS OF DATA FRAMES

The main benefit of a data frame is that it allows you to mix data of different types into a single object in preparation for analysis and modeling. The idea of a data frame is to group data by variables (columns) regardless of their type. Then all the observations on a particular set of variables can be grouped into a single data frame. This is particularly useful in data analysis where it is typical to have a `"character"` variable labeling each observation, one or more `"numeric"` variables of observations, and one or more categorical variables of observations. An example is a built-in data set, `solder`, with information on a welding experiment conducted by AT&T at their Dallas factory.

```
> sampleruns <- sample(row.names(solder),10)
> solder[ sampleruns,]

    Opening Solder Mask PadType Panel skips
380       L  Thick   A3      L7     2     0
545       L  Thick   B3      D4     2     0
462       L   Thin   A3      D6     3     3
809       S  Thick   B6      L9     2     7
609       S  Thick   B3      L4     3    19
492       M   Thin   A6      D6     3     8
525       S   Thin   A6      L6     3    18
313       M   Thin   A3      L6     1     1
408       M  Thick   A6      D7     3    11
540       S   Thin   A6      L9     3    22
```

A sample of 10 of the 900 observations is presented for all six variables. The variable `skips` is the outcome which measures the number of visible soldering skips on a particular run of the experiment. The other variables are categorical and describe the levels of various factors which define the run. The row names on the left are the run numbers for the experiment. Combined in `solder` are character data (the row names), categorical data (the factors), and numeric data (the outcome).

CREATING DATA FRAMES

You can create data frames in several ways:

- `importData` reads data from a variety of application files, as well as from relational databases and ASCII files.
- `read.table` reads in data from an external file.
- `data.frame` binds together S-PLUS objects of various kinds, including existing data frames.
- `as.data.frame` and `data.frame` coerce objects of various types to objects of class `"data.frame"`.

You can also combine existing data frames in several ways, using the `cbind`, `rbind`, and `merge` functions.

The `importData` function is described in detail in the chapter Importing and Exporting Data.

The `read.table` function reads data stored in a text file in table format directly into S-PLUS. The `as.data.frame` function is primarily a support function for the top-level `data.frame` function–it provides a mechanism for defining how new variable classes should be included in newly-constructed data frames. This mechanism is discussed further in section Adding New Classes of Variables to Data Frames (page 123).

For most purposes, when you want to create or modify data frames within S-PLUS, you use the `data.frame` function or one of the combining functions `cbind`, `rbind` or `merge`. This section focuses specifically on the `data.frame` function for combining S-PLUS objects into data frames. The following section discusses the functions for combining existing data frames.

The `data.frame` function is used for creating data frames from existing S-PLUS data objects rather than from data in an external text file. The only required argument to `data.frame` is one or more data objects. All of the objects must produce columns of the same length. Vectors must have the same number of observations as the number of rows of the data frame, matrices must have the same number of rows as the data frame, and lists must have components that match in lengths for vectors or rows for matrices. If the objects don't match appropriately, you get an error message saying the "**arguments imply**

differing number of rows". For example, suppose we have vectors of various modes, each having length 20, along with a matrix with two columns and 20 rows, and a data frame with 20 observations for each of three variables. We can combine these into a data frame as follows.

```
> my.logical <- sample(c(T,F), size=20, replace=T)
> my.complex <- rnorm(20) + runif(20)*1i
> my.numeric <- rnorm(20)
> my.matrix <- matrix(rnorm(40), ncol=2)
> my.df <- kyphosis[1:20, 1:3]
> my.df2 <- data.frame(my.logical, my.complex, my.numeric,
+ my.matrix, my.df)

> my.df2

   my.logical                 my.complex  my.numeric
 1      FALSE -1.8831606111+0.501943978i  1.09345678
 2      FALSE  0.3368386818+0.858758209i  0.09873739
 3       TRUE -0.0003541437+0.381377962i -0.91776485
 4      FALSE  1.2066770747+0.006793533i -1.76152800
 5      FALSE -0.0204049459+0.158040394i  0.30370197
 6      FALSE -1.0119328923+0.860326129i -0.52486689
 7      FALSE  0.9163081264+0.474985190i  1.46745534
 8      FALSE -1.3829848791+0.932033515i  0.45363152
 9      FALSE -0.4695526978+0.795743512i  0.40777969
10       TRUE -0.8035892599+0.256793795i  0.53622210
11       TRUE  0.9026407992+0.637563583i  0.07595690
12       TRUE -1.1558698525+0.655271475i  0.32395563
13      FALSE  0.1049802819+0.706128572i -1.35316648
14       TRUE  0.2302154933+0.373451429i -2.42261503
16      FALSE  2.3956811151+0.086245694i  0.34412995
17       TRUE  0.0824999817+0.258623377i  2.46456956
18      FALSE -0.0248816697+0.417373099i  2.99062594
19       TRUE  0.7525617816+0.636045368i -1.55640891
20       TRUE -1.1078423455+0.011345901i  1.27173450
21       TRUE -2.2280610717+0.517812594i  1.54472022
           X1          X2 Kyphosis Age Number
1  0.80316229  2.28681400   absent  71      3
2 -0.58580658 -0.06509133   absent 158      3
3  0.88756407 -0.89849793  present 128      4
4 -2.35672715  0.68797076   absent   2      5
5  1.26986158 -0.76204606   absent   1      4
```

```
 6 -1.10805175 -1.02164143   absent   1      2
 7  0.56273335  1.34946448   absent  61      2
 8  0.24542337  1.35936982   absent  37      3
 9  0.29190516  2.24852247   absent 113      2
10  0.98675866 -1.27076525  present  59      6
11  0.10125951  0.19835740  present  82      5
12  0.30351481  2.48467422   absent 148      3
13  0.04480753 -1.60470965   absent  18      5
14  1.43504492  1.35172992   absent   1      4
16 -2.45929501 -0.58286780   absent 168      3
17  0.90746053 -0.48598155   absent   1      3
18  0.50886476  0.96350421   absent  78      6
19 -1.11844146 -0.56341008   absent 175      5
20  0.51371598  1.32382209   absent  80      5
21  0.58229738 -0.87364793   absent  27      4
```

The names of the objects are used for the variable names in the data frame. Row names for the data frame are obtained from the first object with a `names`, `dimnames`, or `row.names` attribute having *unique* values. In the above example, the object was `my.df`:

```
> my.df
    Kyphosis Age Number
 1    absent  71      3
 2    absent 158      3
 3   present 128      4
 4    absent   2      5
 5    absent   1      4
 6    absent   1      2
 7    absent  61      2
 8    absent  37      3
 9    absent 113      2
10   present  59      6
11   present  82      5
12    absent 148      3
13    absent  18      5
14    absent   1      4
16    absent 168      3
17    absent   1      3
18    absent  78      6
19    absent 175      5
20    absent  80      5
21    absent  27      4
```

The row names are *not* just the row numbers–in our subset, the number 15 is missing. The fifteenth row of `kyphosis`, and hence `my.df`, has the row name `"16"`.

The attributes of special types of vectors (such as factors) are not lost when they are combined in a data frame. They can be retrieved by asking for the attributes of the particular variable of interest. More detail is given in the section Lists (page 116).

Each vector adds one variable to the data frame. Matrices and data frames provide as many variables to the new data frame as they have columns or variables, respectively. Lists, because they can be built from virtually any data object, are more complicated–they provide as many variables as all of their components taken together.

When combining objects of different types into a data frame, some objects may be altered somewhat to be more suitable for further analysis. For example, numeric vectors and factors remain unchanged in the data frame. Character vectors, however, are converted to factors before being included in the data frame. The conversion is done because S-PLUS assumes that character data will most commonly be taken to be a categorical variable in any modeling that is to follow. If you want to keep a character vector "as is" in the data frame, pass the vector to `data.frame` wrapped in a call to the `I` function, which returns the vector unchanged but with the added class `"AsIs"`.

For example, we can combine the character vector `state.name` as is with a numeric vector in a data frame as follows:

```
> my.df <- data.frame(a=rnorm(50), b=I(state.name))
> my.df[1:4,]
              a        b
1  0.008629243  Alabama
2 -0.038239109   Alaska
3 -1.016802454  Arizona
4 -0.132446253 Arkansas
> mode(my.df$b)

[1] "character"
```

You can provide a character vector as the `row.names` argument to `data.frame` (or another vector which can be converted to character by `as.character`). Just make sure it is the same length as the data objects you are combining into the data frame.

```
> data.frame(price,country,reliab,mileage,type,
+ row.names=c("Acura","Audi","BMW","Chev","Ford",
+ "Mazda","MazdaMX","Nissan","Olds","Toyota"))

       price  country reliab mileage       type
Acura  11950    Japan      5      NA      Small
 Audi  26900  Germany     NA      NA     Medium
. . .
```

COMBINING DATA FRAMES

We have already seen one way to combine data frames–since data frames are legal inputs to the `data.frame` function, you can use `data.frame` directly to combine one or more data frames. For certain specific combinations, other functions may be more appropriate. This section discusses three general cases:

1. Combining data frames *by column*. This case arises when you have new variables to add to an existing data frame, or have two or more data frames having observations of different variables for identical subjects. The principal tool in this case is the `cbind` function. The `data.frame` function could be used in place of the cbind function in the above examples with the same results.
2. Combining data frames *by row*. This case arises when you have multiple studies providing observations of the same variables for different sets of subjects. For this task, use the `rbind` function.
3. Merging (or *joining*) data frames. This case arises when you have two data frames containing some information in common, and you want to get as much information as possible from both data frames about the overlapping cases. For this case, use the `merge` function.

All three of the functions mentioned above (`cbind`, `rbind`, and `merge`) have methods for data frames, but in the usual cases, you can simply call the generic function and obtain the correct result.

For example, `cbind(my.df, newVar)`, `cbind.data.frame()`, and `data.frame()` are all equivalent.

Another way to add one or more columns to an existing data frame is with "`[<-`" or "`[[<-`".

```
> H <- data.frame(a=1:3,b=2:4)
> H[,"c"] <- 4
```

```
> H

  a b c
1 1 2 4
2 2 3 4
3 3 4 4

> H[["d"]] <- 5
> H

  a b c d
1 1 2 4 5
2 2 3 4 5
3 3 4 4 5
```

Combining Data Frames by Column

Suppose you have a data frame consisting of factor variables defining an experimental design. When the experiment is complete, you can add the vector of observed responses as another variable in the data frame. In this case, you are simply adding another column to the existing data frame, and the natural tool for this in S-PLUS is the `cbind` function. For example, consider the simple built-in design matrix `oa.4.2p3`, representing a half-fraction of a `2^4` design.

```
> oa.4.2p3

   A  B  C
1 A1 B1 C1
2 A1 B2 C2
3 A2 B1 C2
4 A2 B2 C1
```

If we run an experiment with this design, we obtain a vector of length four, one observation for each row of the design data frame. We can combine the observations with the design using `cbind` as follows.

```
> run1 <- cbind(oa.4.2p3, resp=c(46, 34, 44, 30))
> run1

   A  B  C resp
1 A1 B1 C1   46
2 A1 B2 C2   34
3 A2 B1 C2   44
4 A2 B2 C1   30
```

Another use of `cbind` is to bind a constant vector to a data frame, as in the following example.

```
> fuel1 <- cbind(1, fuel.frame)
> fuel1
```

```
                  1 Weight Disp. Mileage     Fuel  Type
 Eagle Summit 4 1   2560    97       33 3.030303 Small
  Ford Escort 4 1   2345   114       33 3.030303 Small
 Ford Festiva 4 1   1845    81       37 2.702703 Small
  Honda Civic 4 1   2260    91       32 3.125000 Small
Mazda Protege 4 1   2440   113       32 3.125000 Small
        . . .
```

As a more substantial example, consider the built-in data sets `cu.summary`, `cu.specs`, and `cu.dimensions`. Each of these data sets contains observations about a number of car models, but the list of car models is slightly different in each. All, however, contain data for the cars listed in the data set `common.names`.

```
> common.names
```

```
[1] "Acura Integra"   "Acura Legend"
[3] "Audi 100"        "Audi 80"
[5] "BMW 325i"        "BMW 535i"
[7] "Buick Century"   "Buick Electra"
 . . .
```

The data sets `match.summary`, `match.specs`, and `match.dims` contain the row subscripts to obtain observations about the models listed in `common.names` from, respectively, `cu.summary`, `cu.specs`, and `cu.dimensions`. We can use these data sets and the `cbind` function to compile a general car information data set.

```
> car.mine <- cbind(cu.dimensions[match.dims,],
+ cu.specs[match.specs,], cu.summary[match.summary,],
+ row.names=common.names)
```

Compare `car.mine` to the built-in data set `car.all`, constructed in a similar fashion.

Combining Data Frames by Row

Suppose you are pooling the data from several research studies. You have data frames with observations of equivalent, or roughly equivalent, variables for several sets of subjects. Renaming variables as necessary, you can subscript the data sets to obtain new data sets having a common set of variables. You can then use `rbind` to obtain a new data frame containing all the observations from the studies.

For example, consider the following data frames.

```
> rand.df1

           norm       unif  binom
1   1.64542042 0.45375156     41
2   1.64542042 0.83783769     44
3  -0.13593118 0.31408490     53
4   0.26271524 0.57312325     34
5  -0.01900051 0.25753044     47
6   0.14986005 0.35389326     41
7   0.07429523 0.53649764     43
8  -0.80310861 0.06334192     38
9   0.47110022 0.24843933     44
10 -1.70465453 0.78770638     45

> rand.df2

        norm binom     chisq
1  0.3485193    50 19.359238
2  1.6454204    41 13.547288
3  1.4330907    53  4.968438
4 -0.8531461    55  4.458559
5  0.8741626    47  2.589351
```

These data frames have the common variables `norm` and `binom`; we subscript and combine the resulting data frames as follows.

```
> rbind(rand.df1[,c("norm","binom")],
+ rand.df2[,c("norm", "binom")])
          norm  binom
1   1.64542042    41
2   1.64542042    44
3  -0.13593118    53
4   0.26271524    34
5  -0.01900051    47
6   0.14986005    41
```

```
7   0.07429523    43
8  -0.80310861    38
9   0.47110022    44
10 -1.70465453    45
11  0.34851926    50
12  1.64542042    41
13  1.43309068    53
14 -0.85314606    55
15  0.87416262    47
```

Warning

Use `rbind` (and, in particular, `rbind.data.frame`) only when you have complete data frames, as in the above example. Do not use it in a loop to add one row at a time to an existing data frame–this is very inefficient. To build a data frame, write all the observations to a data file and use `read.table` to read it in.

Merging Data Frames

In many situations, you may have data from multiple sources with some duplicated data. To get the cleanest possible data set for analysis, you want to *merge* or *join* the data before proceeding with the analysis. For example, player statistics extracted from *Total Baseball* overlap somewhat with player statistics extracted from *The Baseball Encyclopedia*. You can use the `merge` function to join two data frames by their common data. For example, consider the following made-up data sets.

```
> baseball.off
     player years.ML    BA HR
1 Whitehead        4 0.308 10
2     Jones        3 0.235 11
3     Smith        5 0.207  4
4   Russell       NA 0.270 19
5      Ayer        7 0.283  5

> baseball.def
     player years.ML   A    FA
1     Smith        5 300 0.974
2     Jones        3   7 0.990
3 Whitehead        4   9 0.980
4    Russell       NA  55 0.963
5       Ayer       7 532 0.955
```

These can be merged by the two columns they have in common using `merge`:

```
> merge(baseball.off, baseball.def)

     player years.ML    BA HR   A    FA
1      Ayer        7 0.283  5 532 0.955
2     Jones        3 0.235 11   7 0.990
3   Russell       NA 0.270 19  55 0.963
4     Smith        5 0.207  4 300 0.974
5 Whitehead        4 0.308 10   9 0.980
```

By default, `merge` joins by the columns having common names in the two data frames. You can specify different combinations using the `by`, `by.x`, and `by.y` arguments. For example, consider the data sets `authors` and `books`.

```
> authors

  FirstName LastName Age  Income         Home
1     Lorne    Green  82 1200000   California
2     Loren     Blye  40   40000   Washington
3     Robin    Green  45   25000   Washington
4     Robin     Howe   2       0      Alberta
5     Billy     Jaye  40   27500   Washington

> books

  AuthorFirstName AuthorLastName        Book
1           Lorne          Green     Bonanza
2           Loren           Blye   Midwifery
3           Loren           Blye   Gardening
4           Loren           Blye  Perennials
5           Robin          Green Who_dun_it?
6            Rich        Calaway       Splus
```

The data sets have different variable names, but overlapping information. Using the `by.x` and `by.y` arguments to merge, we can join the data sets by the first and last names:

```
> merge(authors, books, by.x=c("FirstName", "LastName"),
+ by.y=c("AuthorFirstName", "AuthorLastName"))

  FirstName LastName Age  Income         Home        Book
1     Loren     Blye  40   40000   Washington   Midwifery
2     Loren     Blye  40   40000   Washington   Gardening
3     Loren     Blye  40   40000   Washington  Perennials
4     Lorne    Green  82 1200000   California     Bonanza
5     Robin    Green  45   25000   Washington Who_dun_it?
```

Because the desired "`by`" columns are in the same position in both `books` and `authors`, we can accomplish the same result more simply as follows.

```
> merge(authors, books, by=1:2)
```

More examples can be found in the `merge` help file.

APPLYING FUNCTIONS TO SUBSETS OF A DATA FRAME

To get summaries of variables in a data frame or matrix, use the apply function. For example,

```
> apply(state.x77, 2, mean)
```

where 2 indicates summary by the second dimension (column):

```
Population Income Illiteracy Life Exp Murder HS Grad Frost
Area
    4246.42 4435.8 1.17  70.8786  7.378  53.108 104.46
70735.88
```

For a few common statistical summaries, there are special-purpose functions which perform faster than `apply`, and handle non-numeric columns gracefully, such as `colMeans`, `colSums`, `colVars`, and `colStdevs`. For example, the above example could be replaced by

```
> colMeans(state.x77)
```

Summaries for Variables by Subsets of Rows

A common operation on data with factor variables is to repeat an analysis for each level of a single factor, or for all combinations of levels of several factors. SAS users are familiar with this operation as the **BY** statement. In S-PLUS, you can perform these operations using the `by` or `aggregate` function. Use `aggregate` when you want numeric summaries of each variable computed for each level; use `by` when you want to use all the data to construct a model for each level.

The `aggregate` function allows you to partition a data frame or a matrix by one or more grouping vectors, and then apply a function to the resulting columns. The function must be one that returns a single value, such as `mean` or `sum`. You can also use `aggregate` to partition a time series (univariate or multivariate) by frequency and apply a summary function to the resulting time series.

For data frames, `aggregate` returns a data frame with a factor variable column for each group or level in the index vector, and a column of numeric values resulting from applying the specified function to the subgroups for each variable in the original data frame.

```
> aggregate(state.x77[,c("Population", "Area")],
+       by=state.division,  FUN = sum)

             Group Population   Area
1        New England      12187  62951
2    Middle Atlantic      37269 100318
3     South Atlantic      32946 266909
4 East South Central      13516 178982
5 West South Central      20868 427791
6 East North Central      40945 244101
7 West North Central      16691 507723
8           Mountain       9625 856047
9            Pacific      28274 891972
```

Warning

For most numeric summaries, *all* variables in the data frame must be numeric. Thus, if we attempt to repeat the above example with the kyphosis data, using kyphosis as the `by` variable, we get an error:

```
> aggregate(kyphosis, by=kyphosis$Kyphosis, FUN=sum)

Error in Summary.factor(structure(.Data = c(1, 1, ..:
 A factor is not a numeric object
Dumped
```

Two ways to get summaries in this example are:

```
> aggregate(numerical.matrix(kyphosis), by=kyphosis$Kyphosis, FUN=sum)

> aggregate(kyphosis[,sapply(kyphosis, is.numeric)], # numerical columns
+           by=kyphosis$Kyphosis, FUN=sum)
```

For time series, `aggregate` returns a new, shorter time series that summarizes the values in the time interval given by a new frequency. For instance you can quickly extract the yearly maximum, minimum, and average from the monthly housing start data in the time series `hstart`, as the following examples show.

```
> aggregate(hstart, nf = 1, fun=max)

1966: 143.0 137.0 164.9 159.9 143.8 205.9 231.0 234.2 160.9
start deltat frequency
 1966      1         1

> aggregate(hstart, nf = 1, fun=min)

1966: 62.3 61.7 82.7 85.3 69.2 104.6 150.9 90.6 54.9
start deltat frequency
 1966      1         1

> aggregate(hstart, nf = 1, fun=mean)

1966: 99.6 110.2 128.8 125.0 122.4 173.7 198.2 171.5 112.6
start deltat frequency
 1966      1         1
```

The by function allows you to partition a data frame according to one or more categorical indices (conditioning variables) and then apply a function to the resulting subsets of the data frame. Each subset is considered a separate data frame, hence, unlike the FUN argument to aggregate, the function passed to by does *not* need to have a numeric result. Thus, by is useful for functions that work on data frames by fitting models, for example.

```
> by(kyphosis, INDICES=kyphosis$Kyphosis, FUN=summary)

kyphosis$Kyphosis:absent
  Kyphosis         Age              Number         Start
  absent:64      Min.:  1.00      Min.:2.00      Min.: 1.00
 present: 0   1st Qu.: 18.00   1st Qu.:3.00   1st Qu.:11.00
              Median: 79.00    Median:4.00    Median:14.00
                Mean: 79.89      Mean:3.75      Mean:12.61
             3rd Qu.:131.00   3rd Qu.:5.00   3rd Qu.:16.00
                Max.:206.00      Max.:9.00      Max.:18.00
------------------------------------------------------------
kyphosis$Kyphosis:present
   Kyphosis         Age              Number          Start
  absent: 0      Min.: 15.00      Min.: 3.000      Min.: 1.000
 present:17   1st Qu.: 73.00   1st Qu.: 4.000   1st Qu.: 5.000
              Median:105.00    Median: 5.000    Median: 6.000
                Mean: 97.82      Mean: 5.176      Mean: 7.294
             3rd Qu.:128.00   3rd Qu.: 6.000   3rd Qu.:12.000
                Max.:157.00      Max.:10.000      Max.:14.000
```

The applied function supplied as the `FUN` argument must accept a data frame as its first argument; if you want to apply a function that does not naturally accept a data frame as its first argument, you must define a function that does so on the fly. For example, one common application of the `by` function is to repeat model fitting for each level or combination of levels; the modeling functions, however, generally have a *formula* as their first argument. The following call to `by` shows how to define the `FUN` argument to fit a linear model to each level:

```
> by(kyphosis, list(Kyphosis=kyphosis$Kyphosis,
+       Older=kyphosis$Age>105),
+       function(data)lm(Number~Start,data=data))

Kyphosis:absent
Older:FALSE
Call:
lm(formula = Number~Start, data = data)

Coefficients:
 (Intercept)        Start
    4.885736 -0.08764492
Degrees of freedom: 39 total; 37 residual
Residual standard error: 1.261852

Kyphosis:present
Older:FALSE
Call:
lm(formula = Number~Start, data = data)

Coefficients:
 (Intercept)       Start
    6.371257 -0.1191617
Degrees of freedom: 9 total; 7 residual
Residual standard error: 1.170313

Kyphosis:absent
Older:TRUE
. . .
```

As in the above example, you should define your `FUN` argument simply. If you need additional parameters for the modeling function, specify them fully in the call to the modeling function, rather than attempting to pass them in through a "..." argument.

Warning

Again, as with `aggregate`, you need to be careful that the function you are applying `by` to works with data frames, and often you need to be careful that it works with factors as well. For example, consider the following two examples.

```
> by(kyphosis, kyphosis$Kyphosis, colMeans)

kyphosis$Kyphosis:absent
 Kyphosis      Age Number    Start
       NA 79.89062   3.75 12.60938
------------------------------------------------------------
kyphosis$Kyphosis:present
 Kyphosis      Age   Number    Start
       NA 97.82353 5.176471 7.294118

> by(numerical.matrix(kyphosos, remove=T),
+ kyphosis$Kyphosis,function(data), apply(data,2,max))

Error in FUN(x): Numeric summary undefined for mode
"character"
Dumped
```

The functions `mean` and `max` are not very different, conceptually. Both return a single number summary of their input, both are only meaningful for numeric data. Because of implementation differences, however, the first example returns appropriate values and the second example dumps. However, when all the variables in your data frame are numeric, or when you want to use `by` with a matrix, you should encounter few difficulties.

```
> dimnames(state.x77)[[2]][4] <- "Life.Exp"
> by(state.x77[,c("Murder", "Population", "Life.Exp")],
+ state.region, summary)

INDICES:Northeast
     Murder          Population      Life.Exp
Min.   : 2.400   Min.   :  472   Min.   :70.39
1st Qu.: 3.100   1st Qu.:  931   1st Qu.:70.55
Median : 3.300   Median : 3100   Median :71.23
Mean   : 4.722   Mean   : 5495   Mean   :71.26
3rd Qu.: 5.500   3rd Qu.: 7333   3rd Qu.:71.83
Max.   :10.900   Max.   :18080   Max.   :72.48

INDICES:South
     Murder          Population      Life.Exp
Min.   : 6.20   Min.   :  579   Min.   :67.96
1st Qu.: 9.25   1st Qu.: 2622   1st Qu.:68.98
Median :10.85   Median : 3710   Median :70.07
Mean   :10.58   Mean   : 4208   Mean   :69.71
3rd Qu.:12.27   3rd Qu.: 4944   3rd Qu.:70.33
Max.   :15.10   Max.   :12240   Max.   :71.42
. . .
```

Closely related to the `by` and `aggregate` functions is the `tapply` function, which allows you to partition a *vector* according to one or more categorical indices. Each index is a vector of logical or factor values the same length as the data vector; to use more than one index create a list of index vectors.

For example, suppose you want to compute a mean murder rate by region. You can use `tapply` as follows.

```
> tapply(state.x77[,"Murder"], state.region, mean)
Northeast     South North Central      West
 4.722222 10.58125         5.275 7.215385
```

To compute the mean murder rate by region *and* income, use `tapply`, as the example below illustrates.

```
> income.lev <- cut(state.x77[,"Income"],
+ summary(state.x77[,"Income"])[-4])

> income.lev

 [1]  1  4  3  1  4  4  4  3  4  2  4  2  4  2  3  3  1
[18]  1  1  4  3  3  3 NA  2  2  2  4  2  4  1  4  1  4
[35]  3  1  3  2  3  1  2  1  2  2  1  3  4  1  2  3
attr(, "levels"):
[1] "3098+ thru 3993" "3993+ thru 4519"
[3] "4519+ thru 4814" "4814+ thru 6315"

> tapply(state.x77[,"Murder"],list(state.region,
+ income.lev),mean)

              3098+ thru 3993 3993+ thru 4519
    Northeast         4.10000        4.700000
        South        10.64444       13.050000
North Central              NA        4.800000
         West         9.70000        4.933333
              4519+ thru 4814 4814+ thru 6315
    Northeast            2.85            6.40
        South            7.85            9.60
North Central            5.52            5.85
         West            6.30            8.40
```

ADDING NEW CLASSES OF VARIABLES TO DATA FRAMES

The manner in which objects of a particular data type are included in a data frame is determined by that type's method for the generic function `data.frameAux`. The behavior for most built-in types is derived from one of the six basic cases shown in the table below.

Table 5.1: *Rules for combining objects into data frames.*

Data Types	Sub-types	Rules

`vector`	`numeric`
	`complex`
	`factor`
	`ordered`
	`logical`
		1. contribute a single variable as is

`matrix`	`matrix`	1. each column creates a separate variable. 2. column names used for variable names
`array`		1. used in arrays of three or more dimensions 2. adds as many columns as its last dimension, plus columns indicating the position for the other dimensions.
`character`	`character`	1. contributes as many columns as would a numeric vector, matrix, or array with the same dimensions. 2. each columin the result is converted to a factor
`list`	`list`	1. each component creates one or more separate variables 2. variable names assigned as appropriate for individual components (column names for matrices, etc.)
`model.matrix`	`model.matrix`	1. object becomes a single variable in result
`data.frame`	`data.frame` `design`	1. each variable becomes a variable in result design. 2. variable names used for variable names

If the existing `data.frameAux` methods do not give the desired behavior when you create a new class, you can define your own `data.frameAux` method for the class. In most cases, you can use one of the six paradigm cases, either as is or with slight modifications. For example, the character method is a straightforward modification of the vector method:

```
> data.frameAux.character
function(x, row.names=NULL, optional=F, na.strings="NA",
...)
        data.frameAux.vector(factor(x,exclude =na.strings),
         row.names,optional)
```

This method converts its input to a factor, then calls the function `data.frameAux.vector`.

You can create new methods from scratch, provided they have the same arguments as `data.frameAux`.

```
> data.frameAux
function(x, ...)
UseMethod("data.frameAux")
```

The argument "`...`" allows the generic function to pass any method-specific arguments to the appropriate method.

If you've already built a function to construct data frames from a certain class of data, you can use it in defining your `data.frameAux` method. Your method must return a list (not a data frame), optionally with an attribute `row.names`. For example, if you have an class `myClass` with slots `x`, `y`, and `a`, each vectors of the same length, where `a` contains names, then the following would be suitable:

```
> data.frameAux.myClass
function(x, ...)
{
  y <- list(x=x@x, y=y@y)
  attr(y, "row.names") <- x@a
  y
}
```

Your method must have "x" as the first argument, and may have additional named arguments which are appropriate for your class.

MENU GRAPHICS 6

Introduction **127**
- Overview 128
- General Procedure 129
- Dialogs 130
- Dialog Fields 130
- Graph Options 131

Scatter Plots **132**
- A Basic Example 133
- Line Plots 136
- Grouping Variables 138
- Line Fits 139
- Nonparametric Curve Fits 143
- Multipanel Conditioning 152

Visualizing One-Dimensional Data **157**
- Density Plots 158
- Histograms 162
- QQ Math Plots 164
- Bar Charts 166
- Dot Plots 169
- Pie Charts 171

Visualizing Two-Dimensional Data **174**
- Box Plots 174
- Strip Plots 178
- QQ Plots 180

Visualizing Three-Dimensional Data **183**
- Contour Plots 183
- Level Plots 185
- Surface Plots 187
- Cloud Plots 189

Visualizing Multidimensional Data **191**
Scatterplot Matrices 191
Parallel Plots 194
Multipanel Trellis Graphics 196

Time Series **200**
Line Plots 200
High-Low Plots 204
Stacked Bar Plots 207

References **210**

INTRODUCTION

The power of S-PLUS comes from the integration of its graphics capabilities with its statistical analysis routines. In the Statistics chapter, we show how statistical procedures are performed in S-PLUS. In this chapter, we introduce the S-PLUS graphics that are built into the menu options. It is not necessary to read this entire chapter before you begin generating graphics. Once you've acquired a basic understanding of the way the **Graph** dialogs are organized, you can refer directly to a section of interest.

The dialogs under the **Graph** menu give you access to nearly all of the Trellis functions in S-PLUS: `xyplot`, `densityplot`, `histogram`, `qqmath`, `barchart`, `dotplot`, `piechart`, `bwplot`, `stripplot`, `qq`, `contourplot`, `levelplot`, `wireframe`, `splom`, and `parallel`. Due to the complicated syntax that these functions require, Trellis graphics usually have the steepest learning curve among users. With the graphical user interface, however, you can create highly involved Trellis graphics as easily as you create scatter plots and histograms.

We begin this chapter by presenting general information about the graphics dialogs, and devote the remaining sections to descriptions and examples for each of them. The presentation of the **Scatter Plot** dialog contains the most detail of all the graphics in this chapter. If you are interested in the basic options under the **Titles**, **Axes**, and **Multipanel Conditioning** tabs of the graphics dialogs, see the section Scatter Plots. For all other graphs, we focus on the dialog options specific to particular plot types.

The S-PLUS graphical user interface is designed to create complicated graphs easily and quickly for exploratory data analysis. Not all of the S-PLUS functionality has been built into the menu options, however, and it is therefore necessary to use command line functions in some sections throughout this chapter. For completely customized graphics, you will likely need to resort to the command line functions as well.

Overview

Figure 6.1 displays many elements of the S-PLUS interface.

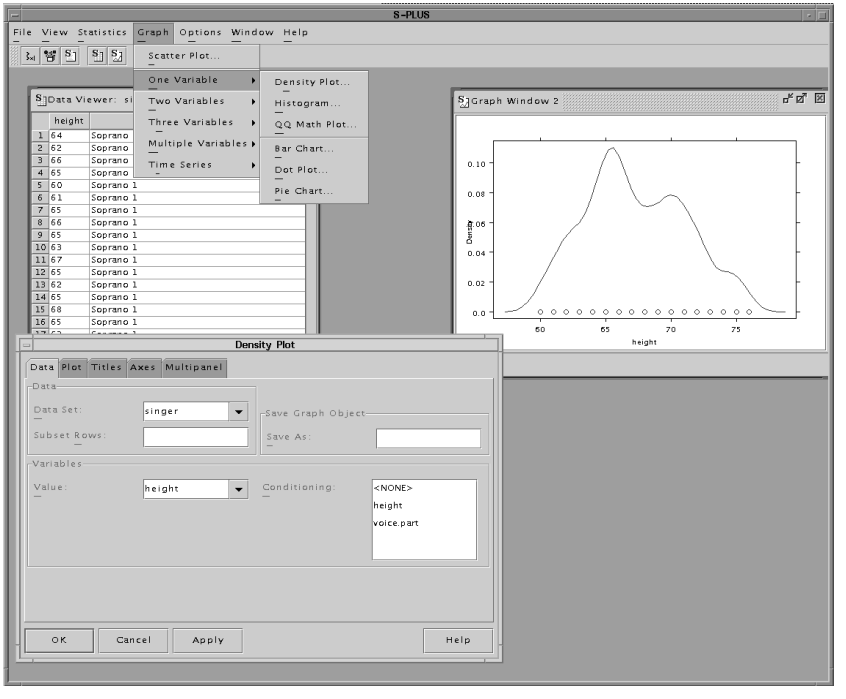


Figure 6.1: *Graphics-related menus and windows.*

- **Graph menu**: The **Graph** menu gives you access to nearly all of the Trellis functions available in S-PLUS. The procedures are logically grouped, with submenus that allow you to precisely specify the procedure you want to use. For example, Figure 6.1 displays the menu tree for density plots. It is selected by choosing **Graph ► One Variable ► Density Plot**.

- **Graph dialogs**: The open dialog in Figure 6.1 is entitled **Density Plot** and is used to display a density estimate for a data set.

- **Data Viewer**: The open window on the left in Figure 6.1 is a **Data** viewer, which you can use to see a data set in its entirety. The **Data** viewer is not a data editor, however, and you cannot use it to modify or create a new data set.

- **Graph Window**: A **Graph** window displays the graphics you create. Figure 6.1 shows the density estimate for a variable in a data set.

- **Commands Window** (not shown): The **Commands** window contains the S-PLUS command line prompt, which you can use to call S-PLUS functions that are not yet implemented in the menu options.

- **Report Window** (not shown): Any error, warning, or informational message generated by a graphics dialog is printed in the **Report** window.

General Procedure

The basic procedure for creating graphs is the same regardless of the type of graph chosen.

1. Choose the graph you want to create from the **Graph** menu. The dialog corresponding to that procedure opens.
2. Select the data set, variables, and options for the procedure you have chosen. (These are slightly different for each dialog.) Click the **OK** or **Apply** button to generate the graph. If you click OK, the dialog closes when the graph is generated; if you click **Apply**, the dialog remains open. We use the **Apply** button extensively in the examples throughout this chapter, as it allows us to experiment with dialog options and build graphs incrementally.
3. Check for messages. If a message is generated, it appears in the **Report** window.
4. Check the result. If everything went well, your graph is displayed in a **Graph** window.

If you want, you can change the variables, parameters, or options in the dialog and click **Apply** to generate new results. S-PLUS makes it easy to experiment with options and to try variations on your analysis.

Dialogs

Much of the graphics functionality in S-PLUS can be accessed through the **Graph** menu. The **Graph** menu includes dialogs for creating one-, two-, and three-dimensional plots, as well as Trellis graphics and time series plots. Many of the dialogs consist of tabbed pages that allow for some formatting, so that you can include legends, titles, and axis labels in your plots. Each dialog has a corresponding function that is executed using dialog inputs as values for function arguments. Usually, it is only necessary to fill in a few fields on the first page of a tabbed dialog to launch the function call.

Dialog Fields

Many dialogs include a **Data Set** field. To specify a data set, you can either type its name directly in the **Data Set** field, or make a selection from the dropdown list. Note that the **Data Set** field recognizes objects of class `"data.frame"` only, and does not accept matrices, vectors, or time series. For this reason, we periodically drop to the **Commands** window in this chapter to create objects that are accepted by the menu options.

Most dialogs that fit statistical models include a **Subset Rows** field that you can use to specify only a portion of a data set. To use a subset of your data in an analysis, enter an S-PLUS expression in the **Subset Rows** field that identifies the rows to use. The expression can evaluate to a vector of logical values: true values indicate which rows to include in the analysis, and false values indicate which rows to drop. Alternatively, the expression can specify a vector of row indices. For example:

- The expression `Species=="bear"` includes only rows for which the `Species` column contains `bear`.
- The expression `Age>=13 & Age<20` includes only rows that correspond to teenage values of the `Age` variable.
- The expression `1:20` includes the first 20 rows of the data.

To use all rows in a data set, leave the **Subset Rows** field blank.

Note that the **Data Set** field recognizes objects of class `"data.frame"` only, and does not accept matrices or vectors. One exception to this is the **Time Series** graphics dialogs, which recognize objects of class `"timeSeries"` only. For this reason, we periodically drop to the **Commands** window in this chapter to create objects that are accepted by the menu options.

Graph Options

The **Options** menu contains a few options that affect the graphics you create from the interface. In particular:

- The **Options ► Dialog Options** window includes a **Create New Graph Window** check box. If this box is selected, as it is by default, then a new **Graph** window is created each time you click **OK** or **Apply**.
- The **Options ► Set Graph Colors** window allows you to select a color scheme for your graphics.
- The **Options ► Graph Options** window governs whether tabbed pages in **Graph** windows are deleted, preserved, or written over when a new plot is generated.

SCATTER PLOTS

The scatter plot is the fundamental visual technique for viewing and exploring relationships in two-dimensional data. In this section, we discuss many of the options available in the **Scatter Plot** dialog, including grouping variables, smoothing, and conditioning. In addition, we also show how you can use the **Scatter Plot** dialog to create one-dimensional line plots of each of your variables. For details on creating line plots specifically for time series data, see the section Time Series.

Creating a scatter plot

From the main menu, choose **Graph ▶ Scatter Plot**. The **Scatter Plot** dialog opens, as shown in Figure 6.2.

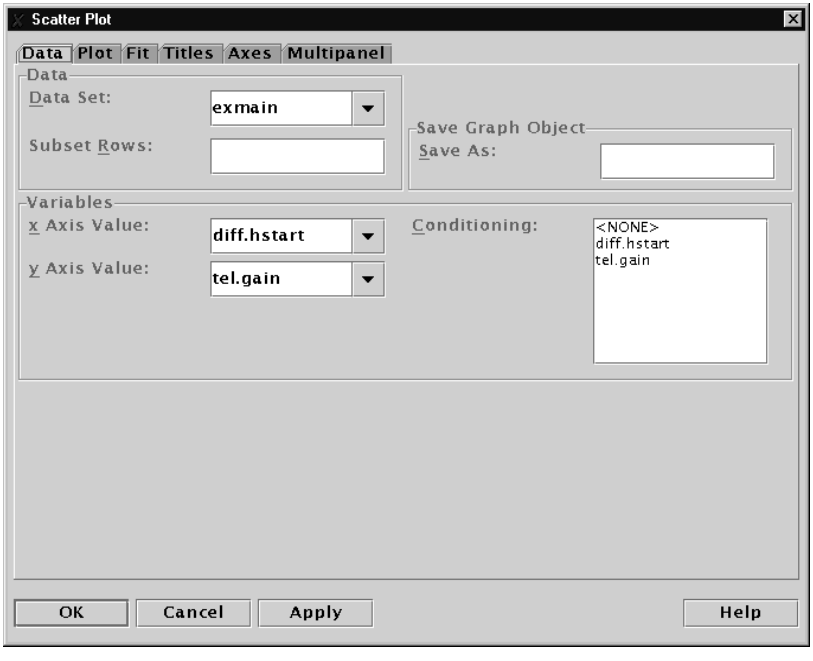


Figure 6.2: *The **Scatter Plot** dialog.*

A Basic Example

The "main gain" data in Table 6.1 present the relationship between the number of housing starts and the number of new main telephone extensions. The observations were recorded once per year on the first of January, for a total of fourteen years beginning in 1971. The first column, "New Housing Starts," is the change in new housing starts from one year to the next in a geographic area around New York City; the units are "sanitized" for confidentiality. The second column, "Gain in Main Residential Telephone Extensions," is the increase in main residential telephone extensions for the same geographic area, again in sanitized units. In this example, we explore the relationship between these two variables using scatter plots.

Table 6.1: *Main gain data.*

New Housing Starts	Gain in Main Residential Telephone Extensions
0.06	1.135
0.13	1.075
0.14	1.496
-0.07	1.611
-0.05	1.654
-0.31	1.573
0.12	1.689
0.23	1.850
-0.05	1.587
-0.03	1.493
0.62	2.049
0.29	1.942
-0.32	1.482
-0.71	1.382

Setting up the data

The data in Table 6.1 are best represented as a data set with two variables. To create this data set, type the following in the **Commands** window:

```
> exmain <- data.frame(
+ diff.hstart = c(0.06, 0.13, 0.14, -0.07, -0.05, -0.31,
+ 0.12, 0.23, -0.05, -0.03, 0.62, 0.29, -0.32, -0.71),
+ tel.gain = c(1.135, 1.075, 1.496, 1.611, 1.654, 1.573,
+ 1.689, 1.850, 1.587, 1.493, 2.049, 1.942, 1.482, 1.382))
> exmain

   diff.hstart tel.gain
1         0.06    1.135
2         0.13    1.075
3         0.14    1.496
4        -0.07    1.611
5        -0.05    1.654
6        -0.31    1.573
7         0.12    1.689
8         0.23    1.850
9        -0.05    1.587
10       -0.03    1.493
11        0.62    2.049
12        0.29    1.942
13       -0.32    1.482
14       -0.71    1.382
```

Exploratory data analysis

If you are responsible for planning the number of new residence extensions that should be installed, you might be interested in whether there is a strong relationship between `diff.hstart` and `tel.gain`. If there is, you can use `diff.hstart` to predict `tel.gain`. As a first step in assessing whether there appears to be a strong relationship between the two variables, make a scatter plot:

1. Open the **Scatter Plot** dialog.
2. Type `exmain` in the **Data Set** field.
3. Select `diff.hstart` as the **x Axis Value** and `tel.gain` as the **y Axis Value**.

4. Click on the **Axes** tab and select **Horizontal** for the **Tick Marks Label Orientation**. This option places horizontal tick labels on both the *x* and *y* axes. By default, labels are parallel to the axes, so that *x* axis tick labels are horizontal and *y* axis labels are vertical.
5. Click **Apply** to leave the dialog open.

The plot is shown in Figure 6.3.

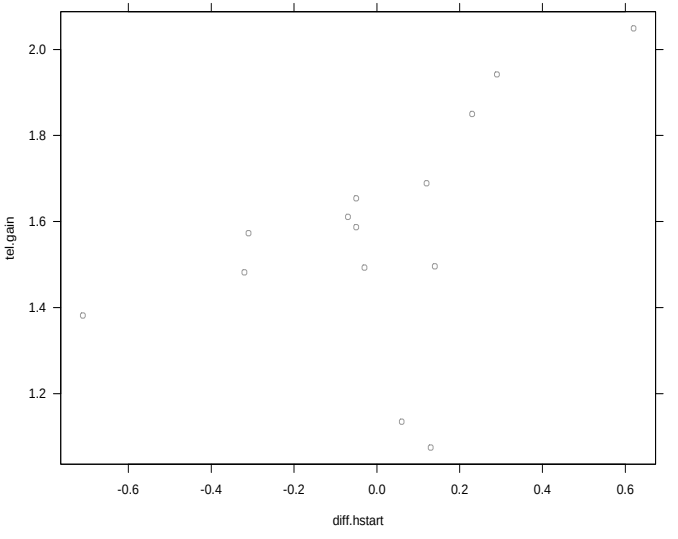


Figure 6.3: *Scatter plot of* `tel.gain` *versus* `diff.hstart`.

The plot immediately reveals two important features in the data. With the exception of two of the data points, there is a positive and roughly linear relationship between new housing starts and the increase in residential telephone extensions. The two exceptional data points are well detached from the remainder of the data; such data points are called *outliers*. In the `exmain` data, the two outliers correspond to the first two observations.

Formatting the graph

You can format a graph with the options in the **Plot**, **Titles**, and **Axes** tabs of the **Scatter Plot** dialog. In the **Plot** tab, you can change the color, style, or size of the plotting symbols and lines. In the **Titles** tab, you can modify axes labels and place a main title on the graph. In the **Axes** tab, you can change the aspect ratio, scale, relation, limits, and tick-label orientation of your axes. For example:

1. Click on the **Plot** tab in the open **Scatter Plot** dialog. Select **Diamond, Solid** as the **Plotting Style**.
2. Click on the **Titles** tab. Type **The Main Gain Data** for the **Main Title**, **New Housing Starts** for the **x Axis Label**, and **Gain in Residential Telephone Extensions** for the **y Axis Label**.
3. Click on the **Axes** tab. Type **-0.9, 0.7** in the **X Limits** field and **0.9, 2.1** in the **Y Limits** field.
4. Click **OK** to close the dialog.

A new **Graph** window appears displaying the changes you made.

Line Plots

Scatter plots are useful tools for visualizing the relationship between any two variables, regardless of whether there is any particular ordering of the *x* axis variable. On the other hand, one of the two variables you want to visualize may be ordered, so that the order in which the observations were taken is as important to the analysis as the values themselves. A line plot, or *index plot*, is a helpful tool for displaying one-dimensional ordered data. In a line plot, the ordered data are plotted along the *y* axis and their corresponding indices are plotted on the *x* axis. This kind of plot arises often in time series data; for details on the line plots available under the **Time Series** graphics menu, see the section Time Series.

Example

In the section A Basic Example on page 133, we created a scatter plot of the variables in the `exmain` data set. In this example, we create a line plot of the `tel.gain` variable.

1. If you have not done so already, create the `exmain` data set with the instructions given on page 134.
2. Open the **Scatter Plot** dialog.
3. Type `exmain` in the **Data Set** field.
4. Select `tel.gain` as the **y Axis Value**. This plots the values in `tel.gain` against a vector of indices that is the same length as `tel.gain`.
5. Click on the **Plot** tab and select **Both Points & Lines** from the **Type** list.
6. Click on the **Titles** tab. Type **index** for the **x Axis Label** and **Gain in Residential Telephone Extensions** for the **y Axis Label**.
7. Click on the **Axes** tab and select **Horizontal** for the **Tick Marks Label Orientation**. This option places horizontal tick labels on both the *x* and *y* axes. By default, labels are parallel to the axes, so that *x* axis tick labels are horizontal and *y* axis labels are vertical.
8. Click **OK**.

The result is shown in Figure 6.4. The fourteen values in `tel.gain`, representing observations made in the years 1971-1984, are plotted sequentially using both points and lines. The observation from 1971 corresponds to the point with the smallest *x* coordinate, and the observation from 1984 corresponds to the point with the largest *x* coordinate. From the plot, we can easily see that gains in new residential telephone extensions were at their lowest during the first two years of the study, rose rapidly in the third year, and then oscillated up and down starting in year 6 of the study.

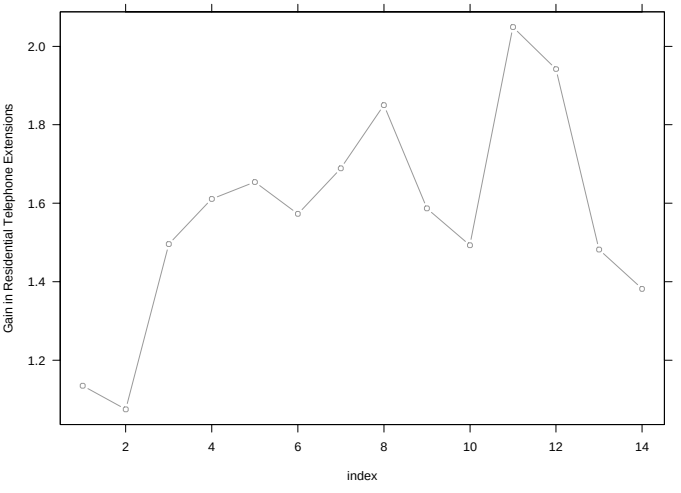


Figure 6.4: *Line plot of* `tel.gain`.

Grouping Variables

It is often useful to plot multiple two-dimensional scatter plots on the same set of axes according to the value of a third factor (categorical) variable. In the **Scatter Plot** dialog, you can choose to vary such scatter plots by symbol color, style, or size. In addition, legends can be included, and are placed on the right side of the graphics area.

Example

The data set `Puromycin` has 23 rows representing the measurement of initial velocity (`vel`) of a biochemical reaction for 6 different concentrations of substrate (`conc`) and two different cell treatments (`state`). In this example, we plot velocity versus concentration with different symbols for the two treatment groups, `treated` and `untreated`.

1. Open the **Scatter Plot** dialog.
2. Type `Puromycin` in the **Data Set** field.
3. Select `conc` as the **x Axis Value** and `vel` as the **y Axis Value**.

4. Click on the **Plot** tab and select `state` as the **Group Variable**. Check the boxes for **Vary Symbol Style** and **Include Legend**.
5. Click on the **Axes** tab and select **Horizontal** for the **Tick Marks Label Orientation**. This option places horizontal tick labels on both the *x* and *y* axes. By default, labels are parallel to the axes, so that *x* axis tick labels are horizontal and *y* axis labels are vertical.
6. Click **OK**.

The result is displayed in Figure 6.5.

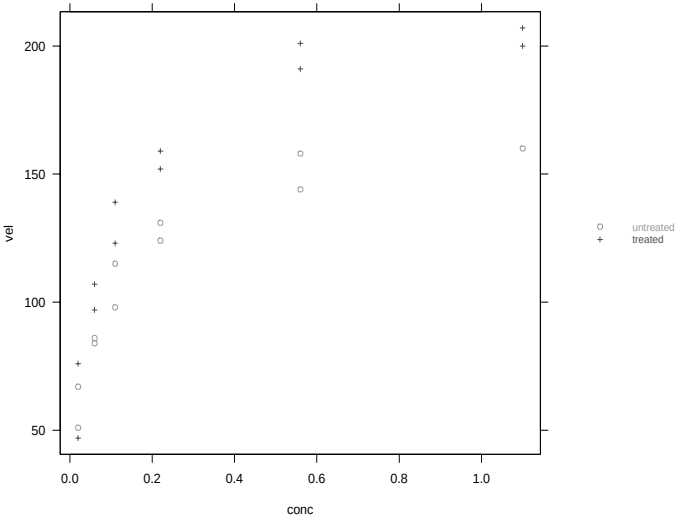


Figure 6.5: *Scatter plot of the* `Puromycin` *data.*

Line Fits

You can fit a straight line to your scatter plot data and superpose the fit with the data. Such a fit helps you visually assess how well the data conforms to a linear relationship between two variables. When the linear fit seems adequate, the fitted straight line plot provides a good visual indication of both the slope of bivariate data, and the variation of the data about the straight line fit. The **Scatter Plot** dialog includes two kinds of line fits in the **Fit** tab, as described below.

- **Linear Least Squares**: computes a line fit via a least squares algorithm.
- **Robust MM**: computes a line fit via a robust fitting criterion. Robust line fits are useful for fitting linear relationships when the random variation in the data is not Gaussian (normal), or when the data contain significant outliers.

Linear Least Squares

The method of *least squares* fits a line to data so that the sum of the squared residuals is minimized. Suppose a set of n observations of the response variable y_i correspond to a set of values of the predictor x_i according to the model $\hat{y} = f(\hat{x})$, where $\hat{y} = (y_1, y_2, \ldots, y_n)$ and $\hat{x} = (x_1, x_2, \ldots, x_n)$. The ith *residual* r_i is defined as the difference between the ith observation y_i and the ith fitted value $\hat{y}_i = \hat{f}(x_i)$: that is, $r_i = y_i - \hat{y}_i$. The method of least squares finds a set of fitted values that minimizes the sum $\sum_{i=1}^{n} r_i^2$.

Example

In the section A Basic Example on page 133, we created a scatter plot of the `exmain` data. You can fit a straight line to the data by the method of least squares and display the result superposed on a scatter plot of the data. The following steps illustrate how to do this.

1. If you have not done so already, create the `exmain` data set with the instructions given on page 134.
2. Open the **Scatter Plot** dialog.
3. Type `exmain` in the **Data Set** field.
4. Select `diff.hstart` as the **x Axis Value** and `tel.gain` as the **y Axis Value**.
5. Click on the **Fit** tab and select **Least Squares** as the **Regression Type**.

6. Click on the **Axes** tab and select **Horizontal** for the **Tick Marks Label Orientation**. This option places horizontal tick labels on both the *x* and *y* axes. By default, labels are parallel to the axes, so that *x* axis tick labels are horizontal and *y* axis labels are vertical.
7. Click **OK**.

The result is shown in Figure 6.6.

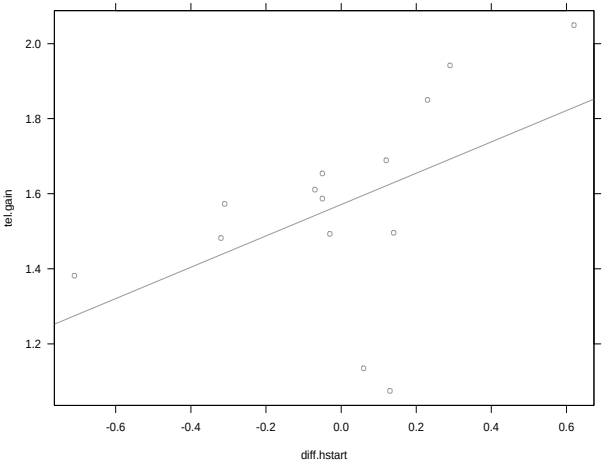


Figure 6.6: *Scatter plot of* `tel.gain` *versus* `diff.hstart` *with a least squares line fit.*

Notice that the two outliers in the data appear to influence the least squares fit by pulling the line downward. This reduces the slope of the line relative to the remainder of the data.

Robust MM

The least squares fit of a straight line is not *robust*, and outliers can have a large influence on the location of the line. A robust method is one that is not significantly influenced by outliers, no matter how large. Robust fitting methods are useful when the random variation in the data is not normal (Gaussian), or when the data contain significant outliers. In such situations, standard least squares may return inaccurate fits. *Robust MM* is one robust fitting method used to guard against outlying observations. The MM method is the robust procedure currently recommended by MathSoft.

Example

In this example, we fit a robust line to the `exmain` data.

1. If you have not done so already, create the `exmain` data set with the instructions given on page 134.
2. Open the **Scatter Plot** dialog.
3. Type `exmain` in the **Data Set** field.
4. Select `diff.hstart` as the **x Axis Value** and `tel.gain` as the **y Axis Value**.
5. Click on the **Fit** tab and select **Robust** as the **Regression Type**.
6. Click on the **Axes** tab and select **Horizontal** for the **Tick Marks Label Orientation**. This option places horizontal tick labels on both the *x* and *y* axes. By default, labels are parallel to the axes, so that *x* axis tick labels are horizontal and *y* axis labels are vertical.
7. Click **OK**.

The result is shown in Figure 6.7.

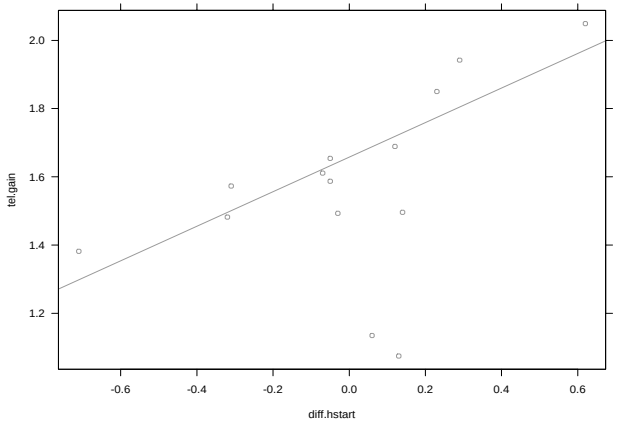


Figure 6.7: *Scatter plot of `tel.gain` versus `diff.hstart` with robust MM line.*

Compare Figure 6.6 to Figure 6.7 and note how much the two outliers influence the least squares line.

Nonparametric Curve Fits

In the previous section, we fit linear parametric functions to scatter plot data. Frequently, you do not have enough prior information to determine what kind of parametric function to use. In such cases, you can fit a *nonparametric curve*, which does not assume a particular type of relationship.

Nonparametric curve fits are also called *smoothers* since they attempt to create a smooth curve showing the general trend in the data. The simplest smoothers use a *running average*, where the fit at a particular x value is calculated as a weighted average of the y values for nearby points. The weight given to each point decreases as the distance between its x value and the x value of interest increases. In the simplest kind of running average smoother, all points within a certain distance (or window) from the point of interest are weighted equally in the average for that point. The window width is called the *bandwidth* of the smoother, and is usually given as a percentage of the total number of data points. Increasing the bandwidth results in a smoother curve fit but may miss rapidly changing features. Decreasing the bandwidth allows the smoother to track rapidly changing features more accurately, but results in a rougher curve fit.

More sophisticated smoothers add variations to the running average approach. For example, smoothly decreasing weights or local linear fits may be used. However, all smoothers have some type of smoothness parameter (bandwidth) controlling the smoothness of the curve. The issue of good bandwidth selection is complicated and has been treated in many statistical research papers. You can, however, gain a good feeling for the practical consequences of varying the bandwidth by experimenting with smoothers on real data.

This section describes how to use four different types of smoothers.

- **Kernel Smoother**: a generalization of running averages in which different weight functions, or *kernels*, may be used. The weight functions provide transitions between points that are smoother than those in the simple running average approach.
- **Loess Smoother**: a noise-reduction approach that is based on local linear or quadratic fits to the data.

- **Spline Smoother**: a technique in which a sequence of polynomials is pieced together to obtain a smooth curve.
- **Supersmoother**: a highly automated variable span smoother. It obtains fitted values by taking weighted combinations of smoothers with varying bandwidths.

In particular, we illustrate how a smoother's bandwidth can be used to control the degree of smoothness in a curve fit.

Kernel Smoothers

A *kernel smoother* is a generalization of running averages in which different weight functions, or *kernels*, may be used. The weight functions provide transitions between points that are smoother than those in the simple running average approach. The default kernel is the *normal* or *Gaussian kernel*, in which the weights decrease with a Gaussian distribution away from the point of interest. Other choices include a triangle, a box, and the Parzen kernel. In a *triangle kernel*, the weights decrease linearly as the distance from the point of interest increases, so that the points on the edge of the smoothing window have a weight near zero. A *box* or *boxcar smoother* weighs each point within the smoothing window equally, and a *Parzen kernel* is a box convolved with a triangle.

Example

The `sensors` data set contains the responses of eight different semiconductor element sensors to varying levels of nitrous oxide (NOx) in a container of air. The engineers who designed these sensors study the relationship between the responses of these eight sensors to determine whether using two sensors, instead of one, allows a more precise measurement of the concentration of NOx. Prior investigation has revealed that there may be a nonlinear relationship between the responses of the two sensors, but not much is known about the details of the relationship. In the examples below, we use kernel smoothers to graphically explore the relationship between the fifth and sixth sensors.

First, create a scatter plot of sensor 5 versus sensor 6 with a box kernel:

1. Open the **Scatter Plot** dialog.
2. Type `sensors` in the **Data Set** field.
3. Select `V5` as the **x Axis Value** and `V6` as the **y Axis Value**.

4. Click on the **Fit** tab. Select **Kernel** as the **Smoothing Type** and **Box** as the **Kernel**.
5. Click on the **Axes** tab and select **Horizontal** for the **Tick Marks Label Orientation**. This option places horizontal tick labels on both the *x* and *y* axes. By default, labels are parallel to the axes, so that *x* axis tick labels are horizontal and *y* axis labels are vertical.
6. Click **Apply** to leave the dialog open.

This results are shown in Figure 6.8.

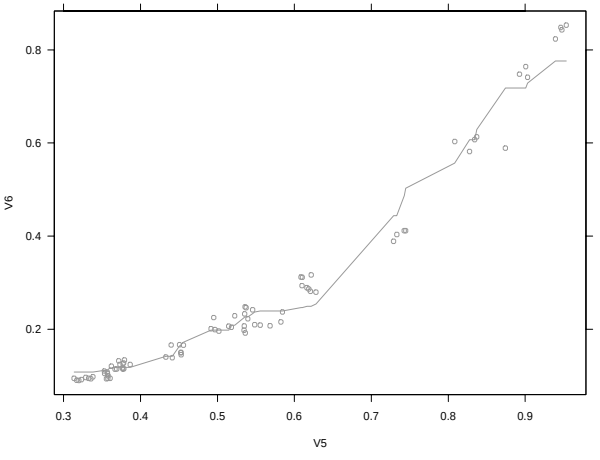


Figure 6.8: *Sensor 5 versus sensor 6 with a box kernel smoother line.*

You can experiment with the smoothing parameter by varying the value in the **Bandwidth** field. For example, click on the **Fit** tab in the open **Scatter Plot** dialog. By default, no bandwidth value is specified. Instead, the standard deviation of the *x* variable is used to compute a good estimate for the bandwidth; this allows the default bandwidth to scale with the magnitude of the data. Type various values between 0.1 and 0.6 in the **Bandwidth** field, clicking **Apply** each time you choose a new value. Each time you click **Apply**, a new **Graph** window appears that displays the updated curve. Note how the smoothness of the fit is affected. Which bandwidth produces the best "eyeball" curve fit? The box kernel smoother with a bandwidth choice of 0.3 is shown in Figure 6.9.

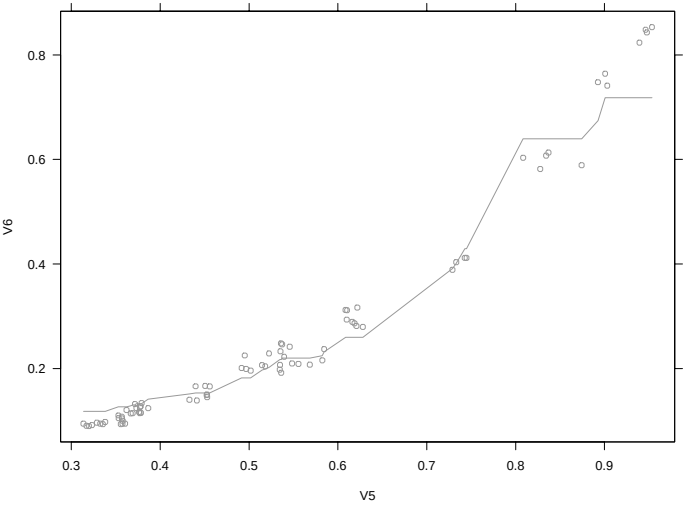


Figure 6.9: *Sensor 5 versus sensor 6 with a box kernel smoother line using a bandwidth of 0.3.*

To obtain a smoother curve, we can experiment with the remaining three kernels. For example, click on the **Fit** tab in the open **Scatter Plot** dialog, choose **Parzen** as the **Kernel**, and click **Apply**. Again, you can also vary the bandwidth choice to see how the smoothness of the fit is affected. Type various values in the **Bandwidth** field, clicking **Apply** each time you choose a new value. Each time you click **Apply**, a new **Graph** window appears that displays the updated curve. The Parzen kernel smoother with a bandwidth choice of 0.15 is shown in Figure 6.10.

When you are finished experimenting, click **OK** to close the dialog.

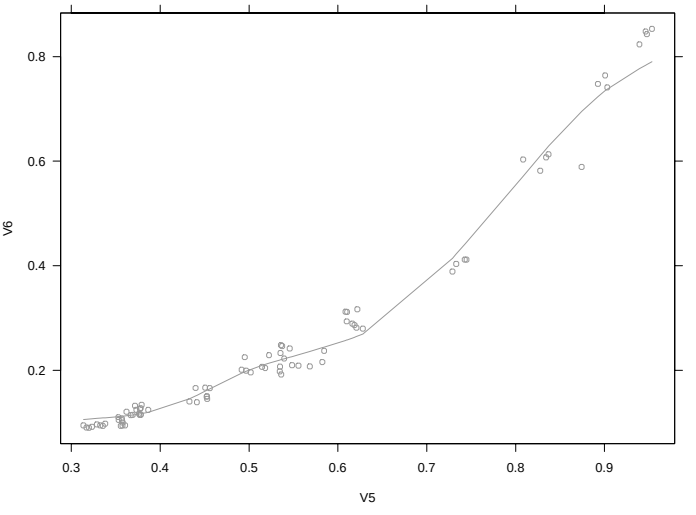


Figure 6.10: *Sensor 5 versus sensor 6 with a Parzen kernel smoother line using a bandwidth of 0.15.*

Loess Smoothers

The *loess smoother*, developed by W.S. Cleveland and others at Bell Laboratories (1979), is a clever approach to smoothing that is essentially a noise-reduction algorithm. It is based on local linear or quadratic fits to the data: at each point, a line or parabola is fit to the points within the smoothing window, and the predicted value is taken as the y value for the point of interest. Weighted least squares is used to compute the line or parabola in each window. Connecting the computed y values results in a smooth curve.

For loess smoothers, the bandwidth is referred to as the *span* of the smoother. The span is a number between 0 and 1, representing the percentage of points that should be included in the fit for a particular smoothing window. Smaller values result in less smoothing, and very small values close to 0 are not recommended. If the span is not specified, an appropriate value is computed using cross-validation. For small samples $(n < 50)$, or if there are substantial serial correlations between observations close in x value, a prespecified fixed span smoother should be used.

Example

In this example, we use loess smoothers to graphically explore the relationship between the fifth and sixth sensors in the `sensors` data set.

1. Open the **Scatter Plot** dialog.
2. Type `sensors` in the **Data Set** field.
3. Select `V5` as the **x Axis Value** and `V6` as the **y Axis Value**.
4. Click on the **Fit** tab and select **Loess** as the **Smoothing Type**.
5. Click on the **Axes** tab and select **Horizontal** for the **Tick Marks Label Orientation**. This option places horizontal tick labels on both the *x* and *y* axes. By default, labels are parallel to the axes, so that *x* axis tick labels are horizontal and *y* axis labels are vertical.
6. Click **Apply** to leave the dialog open.

The result is shown in Figure 6.11.

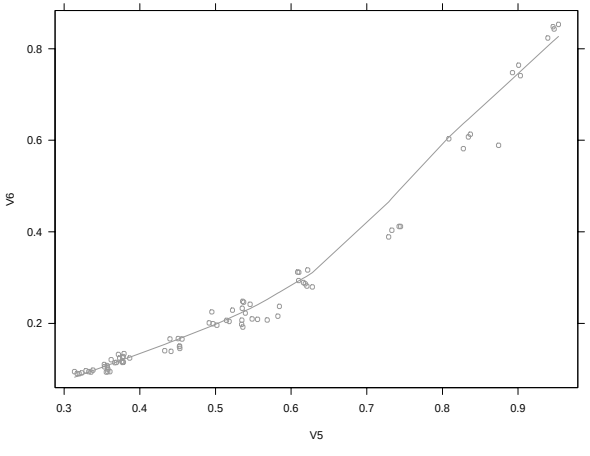


Figure 6.11: *Sensor 5 versus sensor 6 with a loess smoother line.*

You can experiment with the smoothing parameter by varying the value in the **Span** field. For example, click on the **Fit** tab in the open **Scatter Plot** dialog. The bandwidth used to create Figure 6.11 is the default value of 0.75. Since the `sensors` data set has eighty

observations, this means that $0.75 \times 80 = 60$ values are included in the calculation at each point. Type various values between 0.1 and 1 in the **Span** field, clicking **Apply** each time you choose a new value. Each time you click **Apply**, a new **Graph** window appears that displays the updated curve. Note how the smoothness of the fit is affected.

You can also experiment with the degree of the polynomial that is used in the local fit at each point. If you select **Two** as the **Degree** in the **Fit** tab, local quadratic fits are used instead of local linear fits. The **Family** field in the **Fit** tab governs the assumed distribution of the errors in the smoothed curve. The default family is **Symmetric**, which combines local fitting with a robustness feature that guards against distortion by outliers. The **Gaussian** option employs strictly local fitting methods, and can be affected by large outliers. When you are finished experimenting, click **OK** to close the dialog.

Spline Smoothers

Spline smoothers are computed by piecing together a sequence of polynomials. Cubic splines are the most widely used in this class of smoothers, and involve locally cubic polynomials. The local polynomials are computed by minimizing a penalized residual sum of squares. Smoothness is assured by having the value, slope, and curvature of neighboring polynomials match at the points where they meet. Connecting the polynomials results in a smooth fit to the data. The more accurately a smoothing spline fits the data values, the rougher the curve, and vice versa.

The smoothing parameter for splines is called the *degrees of freedom.* The degrees of freedom controls the amount of curvature in the fit, and corresponds to the degree of the local polynomials. The lower the degrees of freedom, the smoother the curve. The degrees of freedom automatically determines the smoothing window, by governing the trade-off between smoothness of the fit and fidelity to the data values. For n data points, the degrees of freedom should be between 1 and $n - 1$. Specifying $n - 1$ degrees of freedom results in a curve that passes through each of the data points exactly.

Example

In this example, we use spline smoothers to graphically explore the relationship between the fifth and sixth sensors in the `sensor` data set.

1. Open the **Scatter Plot** dialog.
2. Type `sensors` in the **Data Set** field.
3. Select `V5` as the **x Axis Value** and `V6` as the **y Axis Value**.
4. Click on the **Fit** tab and select **Smoothing Spline** as the **Smoothing Type**.
5. Click on the **Axes** tab and select **Horizontal** for the **Tick Marks Label Orientation**. This option places horizontal tick labels on both the *x* and *y* axes. By default, labels are parallel to the axes, so that *x* axis tick labels are horizontal and *y* axis labels are vertical.
6. Click **Apply** to leave the dialog open.

You can experiment with the smoothing parameter by varying the value in the **Degrees of Freedom** field. For example, click on the **Fit** tab in the open **Scatter Plot** dialog. The degrees of freedom is set to 3 by default, which corresponds to cubic splines. The `sensors` data set has eighty observations, so type various integer values between 1 and 79 in the **Degrees of Freedom** field (or select values from the drop-down list). If **Crossvalidate** is selected as the **Degrees of Freedom**, the smoothing parameter is computed internally by cross-validation. Click **Apply** each time you choose a new value, and a new **Graph** window appears that displays the updated curve. Note how the smoothness of the fit is affected. When you are finished experimenting, click **OK** to close the dialog.

The spline smoother with 6 degrees of freedom is shown in Figure 6.12.

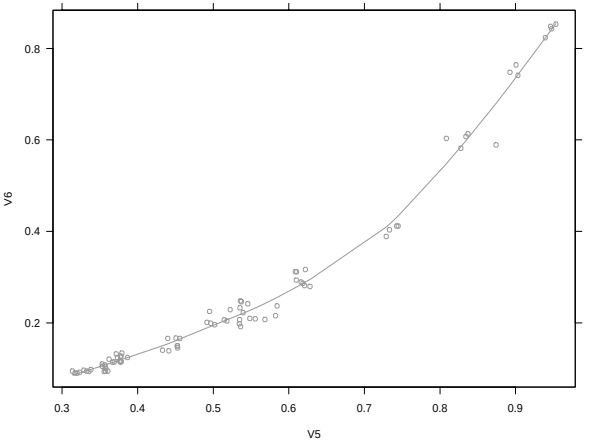


Figure 6.12: *Sensor 5 versus sensor 6 with a spline smoother line using 6 degrees of freedom.*

Friedman's Supersmoother

The *supersmoother* is a highly automated variable span smoother. It obtains fitted values by taking a weighted combination of smoothers with varying bandwidths. Like loess smoothers, the main parameter for supersmoothers is called the *span*. The span is a number between 0 and 1, representing the percentage of points that should be included in the fit for a particular smoothing window. Smaller values result in less smoothing, and very small values close to 0 are not recommended. If the span is not specified, an appropriate value is computed using cross-validation. For small samples $(n < 50)$, or if there are substantial serial correlations between observations close in x value, a prespecified fixed span smoother should be used.

Example

In this example, we use a supersmoother to graphically explore the relationship between the fifth and sixth sensors in the `sensors` data set.

1. Open the **Scatter Plot** dialog.
2. Type `sensors` in the **Data Set** field.
3. Select `V5` as the **x Axis Value** and `V6` as the **y Axis Value**.
4. Click on the **Fit** tab and select **Supersmoother** as the **Smoothing Type**.
5. Click **Apply** to leave the dialog open.

A **Graph** window is created containing a plot. As in the previous examples, you can experiment with the smoothing parameter by varying the value in the **Span** field. For example, click on the **Fit** tab in the open **Scatter Plot** dialog. By default, no span value is specified, so it is computed internally by cross-validation. Type various values between 0.1 and 1 in the **Span** field, clicking **Apply** each time you choose a new value. Each time you click **Apply**, a new **Graph** window appears that displays the updated curve. Note how the smoothness of the fit is affected. When you are finished experimenting, click **OK** to close the dialog.

Multipanel Conditioning

In the section Grouping Variables, we plotted multiple two-dimensional scatter plots on the same set of axes according to the value of a third factor (categorical) variable. It is also possible to place multiple scatter plots on separate axes, conditioned on the value of a third variable. When a conditioning variable is categorical, S-PLUS generates plots for each level. When a conditioning variable is numeric, conditioning is automatically carried out on the sorted unique values; each plot represents either an equal number of observations or an equal range of values.

The **Scatter Plot** dialog, as well as many other dialogs in the **Graph** menu, includes options for specifying conditioning variables, arranging the plots, and labeling the panels. For additional detailed examples on conditioning in the **Graph** dialogs, see the section Visualizing Multidimensional Data.

Example 1

The `ethanol` data set records 88 measurements from an experiment in which ethanol was burned in a single cylinder automobile test engine. The three variables in the experiment are the concentration of nitric oxide and nitrogen dioxide in the engine's exhaust (`NOx`), the compression ratio of the engine (`C`), and the equivalence ratio at which the engine was run (`E`). In this example, we examine the relationship between `NOx` and `E` for various values of `C`.

The conditioning variable `C` is numeric and has 6 unique values: 7.5, 9.0, 12.0, 15.0, and 18.0. We create scatter plots of `NOx` versus `E` for each value of these values. S-PLUS displays the conditioned plots, or *panels*, in the same order that the `levels` function returns the values of the conditioning variable. The effect is the same if we declare the conditioning variable to be a factor directly:

```
> ethanol.fac <- factor(ethanol$C)
> levels(ethanol.fac)

[1] "7.5" "9"   "12"  "15"  "18"
```

In the multipanel graph, the individual scatter plots are therefore placed in order from `C=7.5` to `C=18`.

By default, S-PLUS displays the individual scatter plots in succession from the bottom left corner of the **Graph** window to the top right corner. Figure 6.13 displays the plots generated by the steps below. The scatter plot for `C=7.5` is in the lower left corner of the window, the plot for `C=9.0` is to the right of it, etc.

1. Open the **Scatter Plot** dialog.
2. Type `ethanol` in the **Data Set** field.
3. Select `E` as the **x Axis Value** and `NOx` as the **y Axis Value**. Highlight `C` in the **Conditioning** box.
4. Click on the **Axes** tab. Set the **Aspect Ratio** to be a **Specified Value** and type 0.5 for the **Ratio Value**.
5. Select **Horizontal** for the **Tick Marks Label Orientation**. This option places horizontal tick labels on both the *x* and *y* axes. By default, labels are parallel to the axes, so that *x* axis tick labels are horizontal and *y* axis labels are vertical.
6. Click on the **Multipanel** tab. Select **Unique Values** as the **Interval Type**, and click **Apply** to leave the dialog open.

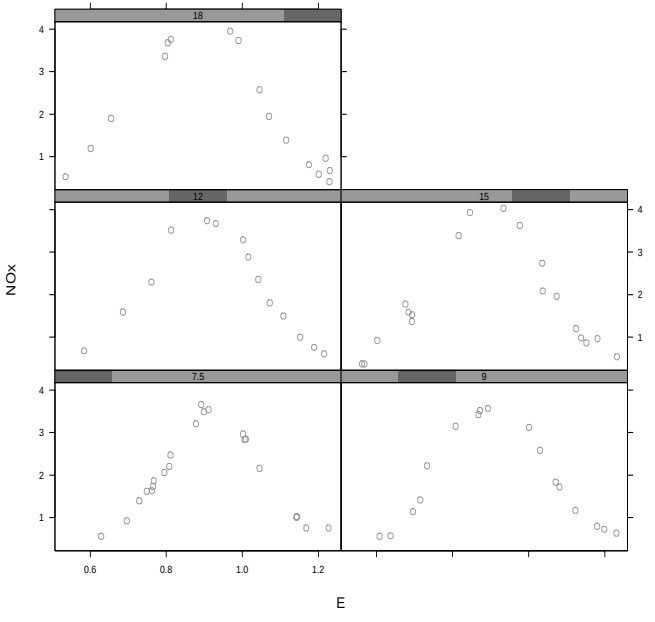


Figure 6.13: *Scatter plots of* `NOx` *versus* `E` *for various values of* `C`.

You can change the layout of the plots in the **Graph** window with the options in the **Multipanel** tab of the open **Scatter Plot** dialog. For example, to start the individual plots in the upper left corner of the window instead of the lower left corner, select **Table Order** from the **Panel Order** list. This places the plot for `C=7.5` in the upper left corner, the plot for `C=9.0` to the right of it, and so on. You can also specify the number of rows and columns in the layout, and the number of pages is computed accordingly. Conversely, you can specify the number of pages, and the panels are placed in appropriate rows and columns. When you are finished experimenting, click **OK** to close the dialog.

Example 2

In this example, we examine the relationship between NOx and C for various values of E. However, E varies in a nearly continuous way: there are 83 unique values out of 88 observations. Since E is a continuous variable, each panel represents either an equal number of observations or an equal range of values.

1. Open the **Scatter Plot** dialog.
2. Type ethanol in the **Data Set** field.
3. Select C as the **x Axis Value** and NOx as the **y Axis Value**. Highlight E in the **Conditioning** box.
4. Click on the **Axes** tab. Set the **Aspect Ratio** to be **Bank to 45 Degree**.
5. Select **Horizontal** for the **Tick Marks Label Orientation**. This option places horizontal tick labels on both the *x* and *y* axes. By default, labels are parallel to the axes, so that *x* axis tick labels are horizontal and *y* axis labels are vertical.
6. Suppose we want to generate a 2×5 grid containing 9 scatter plots, with an equal number of observations in each panel. Click on the **Multipanel** tab. Type 5 for the **# of Columns** and 2 for the **# of Rows**. Type 9 in the **# of Panels** field and 0.25 as the **Overlap Fraction**.
7. Click **Apply** to leave the dialog open.

The result is displayed in Figure 6.14. Since the **Panel Order** is set to **Graph Order** by default, the minimum values of E are in the lower left panel and the maximum values are in the upper right panel. To place the plot with the minimum values in the upper left corner of the window instead, click on the **Multipanel** tab in the open **Scatter Plot** dialog and select **Table Order** as the **Panel Order**. To generate plots according to equal-length intervals of the values in E, select **Equal Ranges** as the **Interval Type**.

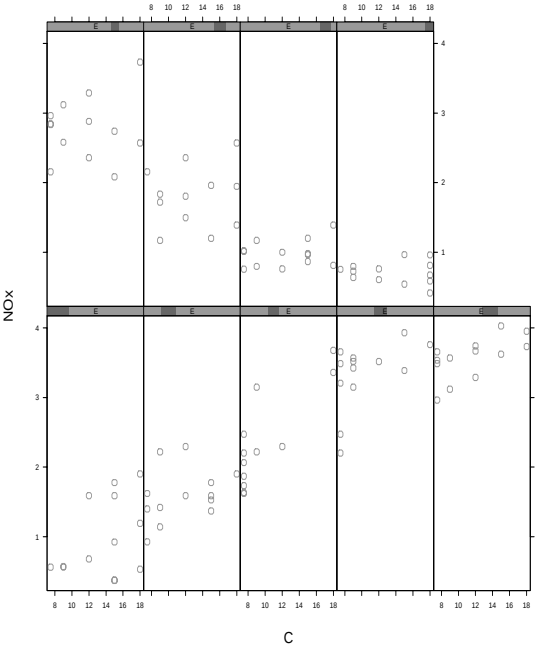


Figure 6.14: *Scatter plots of* `NOx` *versus* `C` *for various values of* `E`.

The **Overlap Fraction** in the **Multipanel Conditioning** tab governs the amount of points that are shared by successive intervals of the conditioning variable. The endpoints of the intervals are chosen to make either the number of points (if **Equal Counts** is chosen) or the length of the intervals (if **Equal Ranges** is chosen) as nearly equal as possible. At the same time, the amount of points shared by successive intervals is kept as close to the **Overlap Fraction** as possible. If the **Overlap Fraction** is between 0 and 1, it is the fraction of points shared between adjacent intervals. If the **Overlap Fraction** is greater than or equal to 1, it is the number of points shared between adjacent intervals.

When you are finished experimenting, click **OK** to close the dialog.

VISUALIZING ONE-DIMENSIONAL DATA

A *one-dimensional data object* is sometimes referred to as a (single) *data sample*, a set of *univariate observations*, or simply a batch of data. In this section, we examine a number of basic plot types useful for exploring a one-dimensional data object.

- **Density Plot**: an estimate of the underlying probability density function for a data set.
- **Histogram**: a display of the number of data points that fall in each of a specified number of intervals. A histogram gives an indication of the relative density of the data points along the horizontal axis.
- **QQ Math Plot**: an extremely powerful tool for determining a good approximation to a data set's distribution. The most common is the *normal probability plot*, or *normal qqplot*, which is used to test whether the distribution of a data set is nearly Gaussian.
- **Bar Chart**: a display of the relative magnitudes of observations in a data set. A bar is plotted for each data point, where the height of a bar is determined by the value of the data point. The **Bar Chart** dialog can also tabulate counts for a factor variable in a data set.
- **Dot Plot**: a tool that displays the same information as a bar chart or pie chart, but in a form that is often easier to grasp.
- **Pie Chart**: a graph that shows the share of individual values in a variable, relative to the sum total of all the values.

These visualization plots are simple but powerful exploratory data analysis tools that can help you quickly grasp the nature of your data. Such an understanding can help you avoid the misuse of statistical inference methods, such as using a method appropriate only for a normal (Gaussian) distribution when the distribution is strongly non-normal.

Density Plots

As a first step in analyzing one-dimensional data, it is often useful to study the shape of its distribution. A *density plot* displays an estimate of the underlying probability density function for a data set, and allows you to approximate the probability that your data fall in any interval.

In S-PLUS, density plots are essentially kernel smoothers. The algorithm used to compute the plots is therefore similar to those presented in the section Nonparametric Curve Fits. A smoothing window is centered on each x value, and the predicted y value in the density plot is calculated as a weighted average of the y values for nearby points. The size of the smoothing window is called the *bandwidth* of the smoother. Increasing the bandwidth results in a smoother curve but may miss rapidly changing features. Decreasing the bandwidth allows the smoother to track rapidly changing features more accurately, but results in a rougher curve fit. The **Density Plot** dialog includes various methods for estimating good bandwidth values.

The weight given to each point in a smoothing window decreases as the distance between its x value and the x value of interest increases. *Kernel functions* specify the way in which the weights decrease: kernel choices for density plots include a cosine curve, a normal (Gaussian) kernel, a rectangle, and a triangle. The default kernel is Gaussian, where the weights decrease with a normal (Gaussian) distribution away from the point of interest. A rectangular kernel weighs each point within the smoothing window equally, and a triangular kernel has linearly decreasing weights. In a cosine kernel, weights decrease with a cosine curve away from the point of interest.

Creating a density plot

From the main menu, choose **Graph ► One Variable ► Density Plot**. The **Density Plot** dialog opens, as shown in Figure 6.15.

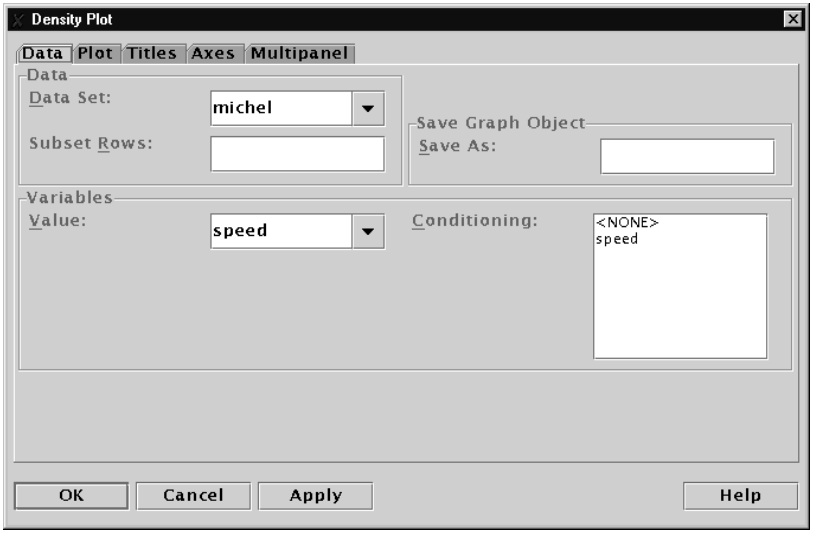


Figure 6.15: *The **Density Plot** dialog.*

Example

In 1876, the French physicist Cornu reported a value of 299,990 km/sec for c, the speed of light. In 1879, the American physicist A.A. Michelson carried out several experiments to verify and improve Cornu's value. Michelson obtained the following 20 measurements of the speed of light:

```
 850  740  900  1070  930  850  950  980  980  880
1000  980  930   650  760  810 1000 1000  960  960
```

To obtain Michelson's actual measurements, add 299,000 km/sec to each of the above values.

The 20 observations can be thought of as observed values of 20 random variables with a common but unknown mean-value location μ. If the experimental setup for measuring the speed of light is free of bias, then it is reasonable to assume that μ is the true speed of light. In evaluating these data, we seek answers to at least four questions, listed below.

1. What is the speed of light μ?
2. Has the speed of light changed relative to our best previous value $\mu_0 = 299,990$ km/sec?
3. What is the uncertainty associated with our answers to (1) and (2)?
4. What is the shape of the distribution of the data?

The first three questions were probably in Michelson's mind when he gathered his data. The last two must be answered to determine which techniques can obtain valid statistical inferences from the data. In this example, we use density plots to graphically analyze the distribution of the Michelson data. In the Statistics chapter, we revisit these data and perform various statistical tests to answer questions (2) and (3).

Setting up the data

We begin analyzing the Michelson data by first creating an S-PLUS data frame that contains it. To do this, type the following in the **Commands** window:

```
> michel <- data.frame(speed = c(850, 740, 900,
+ 1070, 930, 850, 950, 980, 980, 880, 1000, 980,
+ 930, 650, 760, 810, 1000, 1000, 960, 960))
```

Exploratory data analysis

To obtain a useful exploratory view of the Michelson data, create a density plot as follows:

1. Open the **Density Plot** dialog.
2. Type `michel` in the **Data Set** field.
3. Select `speed` as the **Value**.
4. Click **Apply** to leave the dialog open.

The result is shown in Figure 6.16. The rug at the bottom of the density plot shows the unique *x* values in the data set.

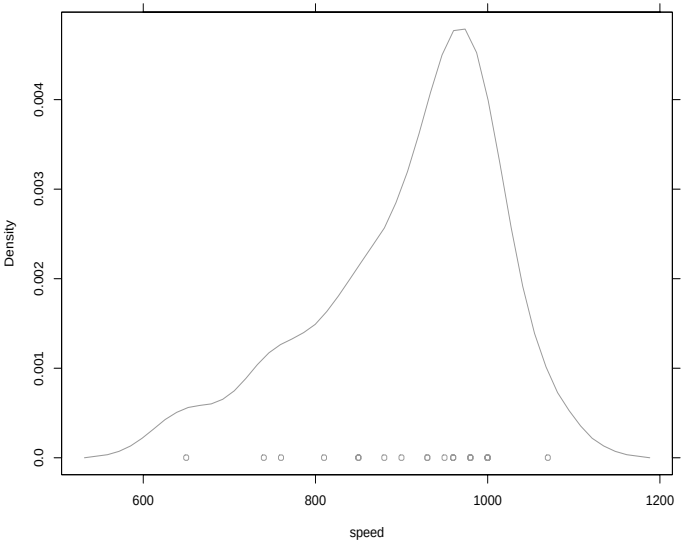


Figure 6.16: *Density estimate of the Michelson data.*

To experiment with the smoothing kernel, click on the **Plot** tab in the open **Density Plot** dialog and choose a new function from the **Window Type** list. The **Number of Points** field specifies the number of equally spaced points at which to estimate the density; the **From** and **To** fields define the range of the equally spaced points. The **Width Method** field specifies the algorithm for computing the width of the smoothing window. Available methods are the histogram bin (**Hist Bin**), normal reference density (**Normal Ref**), biased cross-validation (**Biased CV**), unbiased cross-validation (**Unbiased CV**), and Sheather & Jones pilot estimation of derivatives (**Est Deriv**). You can also define your own window by selecting **Specified Value** from the **Width Method** list, and then typing a number for the **Width Value**. For more information on the methods used to compute the width of a smoothing window, see Venables and Ripley (1999).

When you are finished experimenting, click **OK** to close the dialog.

Histograms

Histograms display the number of data points that fall in each of a specified number of intervals. A histogram gives an indication of the relative density of the data points along the horizontal axis. For this reason, density plots are often superposed with (scaled) histograms.

By default, the **Histogram** dialog displays vertical bars. For details on horizontal bar plots, see the section Bar Charts.

Creating a histogram

From the main menu, choose **Graph ▶ One Variable ▶ Histogram**. The **Histogram** dialog opens, as shown in Figure 6.17.

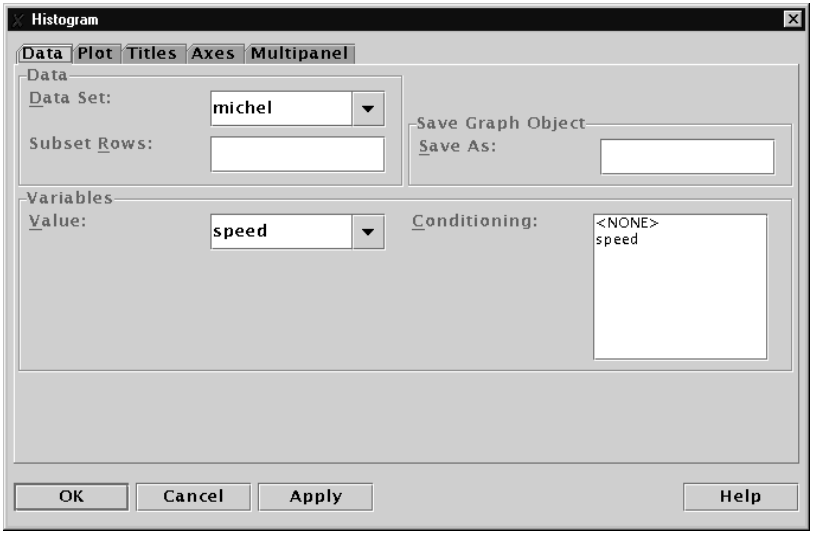


Figure 6.17: *The **Histogram** dialog.*

Example

In the section Density Plots on page 158, we created a probability density estimate for the `michel` data. In this example, we plot a histogram of the data.

1. If you have not done so already, create the `michel` data set with the instructions given on page 160.
2. Open the **Histogram** dialog.
3. Type `michel` in the **Data Set** field and select `speed` as the **Value**.

4. Click on the **Axes** tab and select **Horizontal** for the **Tick Marks Label Orientation**. This option places horizontal tick labels on both the *x* and *y* axes. By default, labels are parallel to the axes, so that *x* axis tick labels are horizontal and *y* axis labels are vertical.
5. Click **Apply** to leave the dialog open.

The result is shown in Figure 6.18.

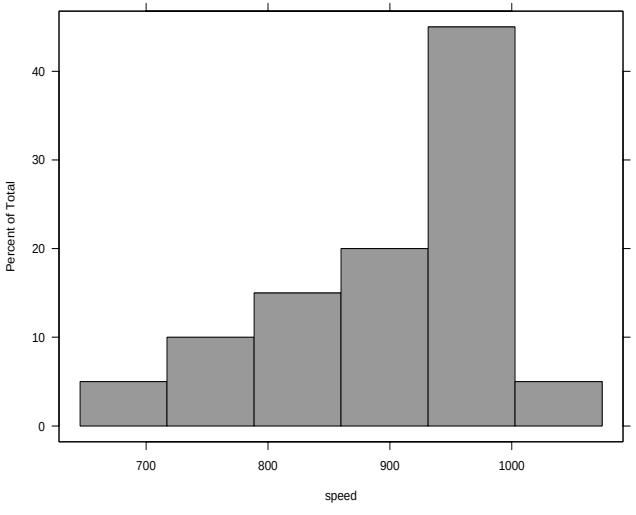


Figure 6.18: *Histogram of the Michelson data.*

By default, S-PLUS displays histograms scaled as probability densities. To display the raw counts in each histogram bin instead, click on the **Plot** tab in the open **Histogram** dialog and select **Count** as the **Bar Height Type**.

S-PLUS computes the number of intervals in a histogram automatically to balance the trade-off between obtaining smoothness and preserving detail. To experiment with the algorithm used to compute the intervals, click on the **Plot** tab in the open **Histogram** dialog. There are three algorithms available in the **Binning Method** list: **Freedman-Diaconi**, **Scott**, and **Sturges**. You can also define your own number of intervals by selecting **Specified Value** from the

Binning Method list, and then typing a number for the **Number of Bins**. For more information on the methods used to compute the number of bins, see Venables and Ripley (1999).

When you are finished experimenting, click **OK** to close the dialog.

QQ Math Plots

The quantile-quantile plot, or *qqplot*, is an extremely powerful tool for determining a good approximation to a data set's distribution. In a qqplot, the ordered data are graphed against quantiles of a known theoretical distribution. If the data points are drawn from the theoretical distribution, the resulting plot is close to a straight line in shape. The most common in this class of one-dimensional plots is the *normal probability plot*, or *normal qqplot*, which is used to test whether the distribution of a data set is nearly normal (Gaussian).

Creating a QQ math plot

From the main menu, choose **Graph ▶ One Variable ▶ QQ Math Plot**. The **QQ Math Plot** dialog opens, as shown in Figure 6.19.

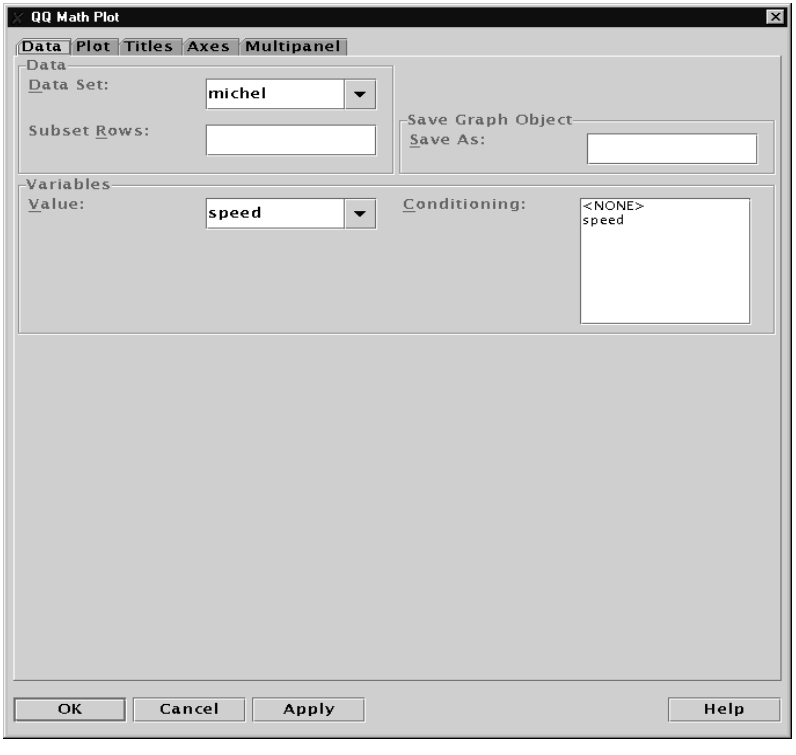


Figure 6.19: *The **QQ Math Plot** dialog.*

Example

In the section Density Plots on page 158, we created a probability density estimate for the `michel` data. In this example, we compare the data to a normal (Gaussian) distribution.

1. If you have not done so already, create the `michel` data set with the instructions given on page 160.
2. Open the **QQ Math Plot** dialog.
3. Type `michel` in the **Data Set** field and select `speed` as the **Value**.
4. Click **Apply** to leave the dialog open.

The result is shown in Figure 6.20.

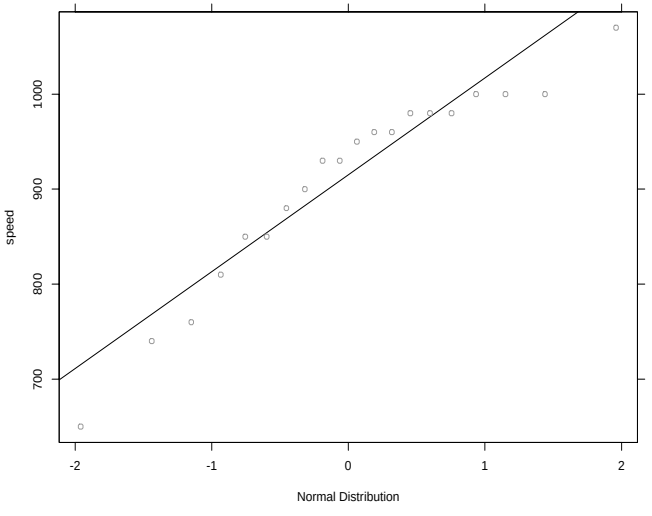


Figure 6.20: *Normal QQ plot for the Michelson data.*

By default, S-PLUS includes a reference line in qqplots. To omit the line from a graph, deselect the **Include Reference Line** option in the **Plot** page of the dialog.

The points in Figure 6.20 do not fall particularly close to a straight line, which suggests that the data may not be normally distributed. You can experiment with the chosen theoretical distribution by varying the selection in the **Distribution** list. For example, click on

the **Plot** tab in the open **QQ Math Plot** dialog. By default, the **Distribution** is **normal** with a **Mean** of 0 and a **Std. Deviation** of 1. Select **t** as the **Distribution**, type 5 in the **Deg of Freedom 1** box, and click **Apply**. Does the *t* distribution with 5 degrees of freedom produce a more linear qqplot? When you are finished experimenting, click **OK** to close the dialog.

Bar Charts

A *bar chart* displays a bar for each point in a set of observations, where the height of a bar is determined by the value of the data point. The **Bar Chart** dialog also contains an option for tabulating the values in your data set according to the levels of a categorical variable. This allows you to view a count of the observations that are associated with each level of a factor variable.

By default, S-PLUS generates horizontal bar charts from the menu options. If you require vertical bar charts, you should use the command line function `barplot`.

Creating a bar chart

From the main menu, choose **Graph ► One Variable ► Bar Chart**. The **Bar Chart** dialog opens, as shown in Figure 6.21.

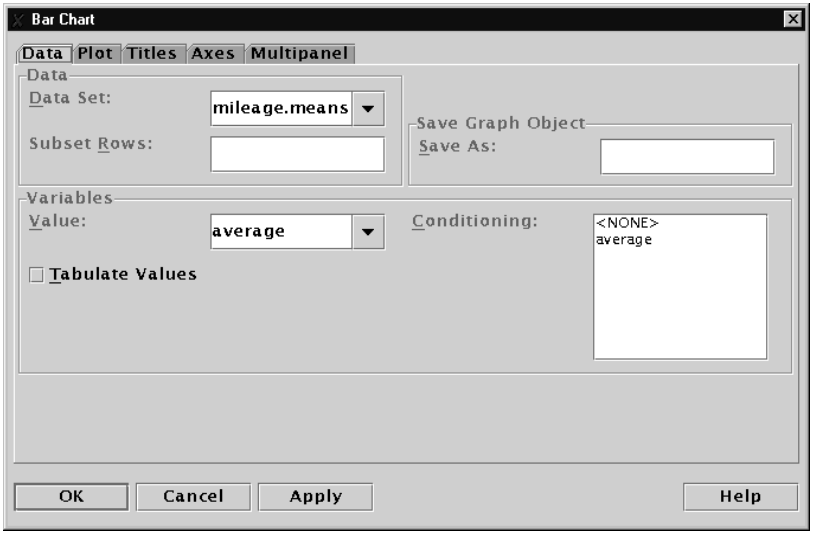


Figure 6.21: *The **Bar Chart** dialog.*

Example 1

The data set `fuel.frame` is taken from the April 1990 issue of *Consumer Reports*. It contains 60 observations (rows) and 5 variables (columns). Observations of weight, engine displacement, mileage, type, and fuel were taken for each of sixty cars. In this example, we graphically analyze the average mileage for each of the six types of cars.

To create a `mileage.means` data set containing the average `Mileage` for each `Type` of car, type the following in the **Commands** window:

```
> mileage.means <-data.frame(average =
+ tapply(fuel.frame$Mileage, fuel.frame$Type, FUN=mean))
> mileage.means

         average
Compact 24.13333
  Large 20.33333
 Medium 21.76923
  Small 31.00000
 Sporty 26.00000
    Van 18.85714
```

Create a bar chart of the `mileage.means` data as follows:

1. Open the **Bar Chart** dialog.
2. Type `mileage.means` in the **Data Set** field.
3. Select `average` as the **Value**. Deselect the **Tabulate Values** option.
4. Click on the **Titles** tab and type `mileage.means` for the **X Axis Label**.
5. Click **OK**.

The horizontal bar chart is shown in Figure 6.22. Note that the bars in the chart are placed according to the order in the data set: `Compact`, the first element in `mileage.means`, appears with the smallest y value in the chart, and `Van`, the last element in `mileage.means`, appears with the largest y value.

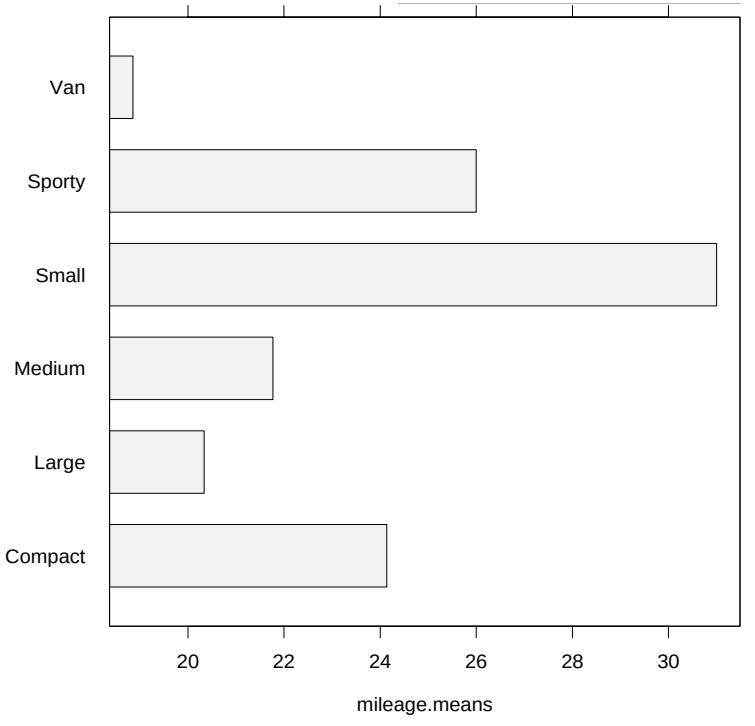


Figure 6.22: *A bar chart of average mileage in the* `fuel.frame` *data set.*

Example 2

In this example, we tabulate the number of cars in the `fuel.frame` data set for each level of the `Type` factor variable.

1. Open the **Bar Chart** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. Select `Type` as the **Value**.
4. Verify that the **Tabulate Values** option is checked.
5. Click **OK**.

A **Graph** window appears that displays a bar chart of the tabulated values in `fuel.frame`. Note that the bars in the chart are placed according to the levels in the `Type` variable: `Compact`, the first level `Type`, appears with the smallest *y* value in the chart, and `Van`, the last

level in `Type`, appears with the largest *y* value. You can view the order of the levels in a factor variable by using the `levels` function in the **Commands** window:

```
> levels(fuel.frame$Type)

[1] "Compact" "Large"   "Medium"  "Small"   "Sporty"  "Van"
```

Dot Plots

The dot plot was first described by Cleveland in 1985 as an alternative to bar charts and pie charts. The dot plot displays the same information as a bar chart or pie chart, but in a form that is often easier to grasp. Instead of bars or pie wedges, dots and gridlines are used to mark the data values in dot plots. In particular, the dot plot reduces most data comparisons to straightforward length comparisons on a common scale.

Creating a dot plot

From the main menu, choose **Graph ▶ One Variable ▶ Dot Plot**. The **Dot Plot** dialog opens, as shown in Figure 6.23.

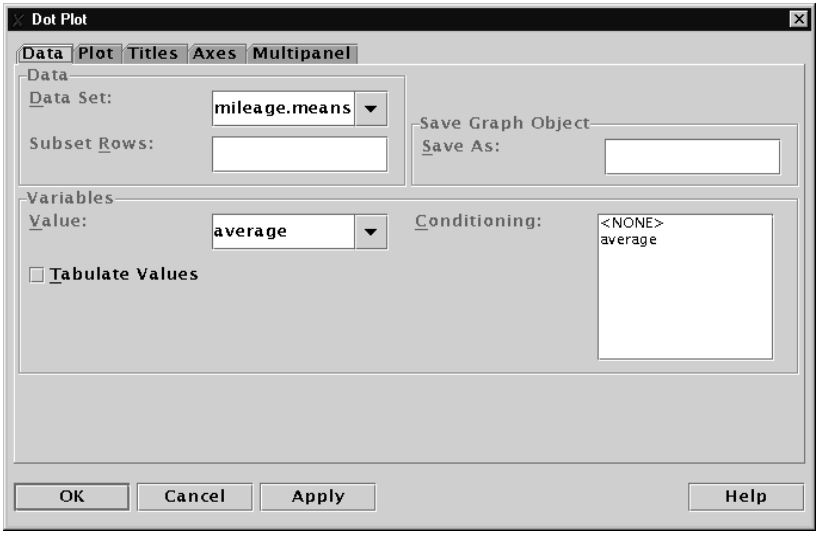


Figure 6.23: *The **Dot Plot** dialog.*

Example 1

In the section Bar Charts on page 166, we used bar charts to graphically display the `mileage.means` data set. In this example, we create a dot plot of these data.

1. If you have not done so already, create the `mileage.means` data set with the instructions given on page 167.
2. Open the **Dot Plot** dialog.
3. Type `mileage.means` in the **Data Set** field.
4. Select `average` as the **Value**. Deselect the **Tabulate Values** option.
5. Click on the **Titles** tab and type `mileage.means` for the **X Axis Label**.
6. Click **OK**.

The result is shown in Figure 6.24. Note that the plot labels are placed according to the order in the data set: `Compact`, the first element in `mileage.means`, appears with the smallest *y* value in the plot, and `Van`, the last element in `mileage.means`, appears with the largest *y* value.

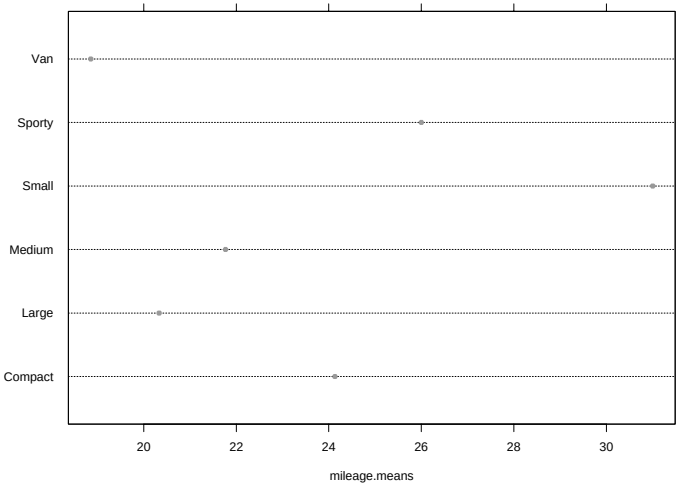


Figure 6.24: *Dot plot of average mileage in the `fuel.frame` data set.*

Example 2

In this example, we tabulate the number of cars in the `fuel.frame` data set for each level of the `Type` factor variable.

1. Open the **Dot Plot** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. Select `Type` as the **Value**.
4. Verify that the **Tabulate Values** option is checked.
5. Click **OK**.

A **Graph** window appears that displays a dot plot of the tabulated values in `fuel.frame`. Note that the plot labels are placed according to the levels in the `Type` variable: `Compact`, the first level `Type`, appears with the smallest *y* value in the chart, and `Van`, the last level in `Type`, appears with the largest *y* value. You can view the order of the levels in a factor variable by using the `levels` function in the **Commands** window:

```
> levels(fuel.frame$Type)

[1] "Compact" "Large"   "Medium"  "Small"   "Sporty"  "Van"
```

Pie Charts

A *pie chart* shows the share of individual values in a variable, relative to the sum total of all the values. Pie charts display the same information as bar charts and dot plots, but can be more difficult to interpret. This is because the size of a pie wedge is relative to a sum, and does not directly reflect the magnitude of the data value. Because of this, pie charts are most useful when the emphasis is on an individual item's relation to the whole; in these cases, the sizes of the pie wedges are naturally interpreted as percentages. When such an emphasis is not the primary point of the graphic, a bar chart or a dot plot is preferred.

Creating a pie chart

From the main menu, choose **Graph ▶ One Variable ▶ Pie Chart**. The **Pie Chart** dialog opens, as shown in Figure 6.25.

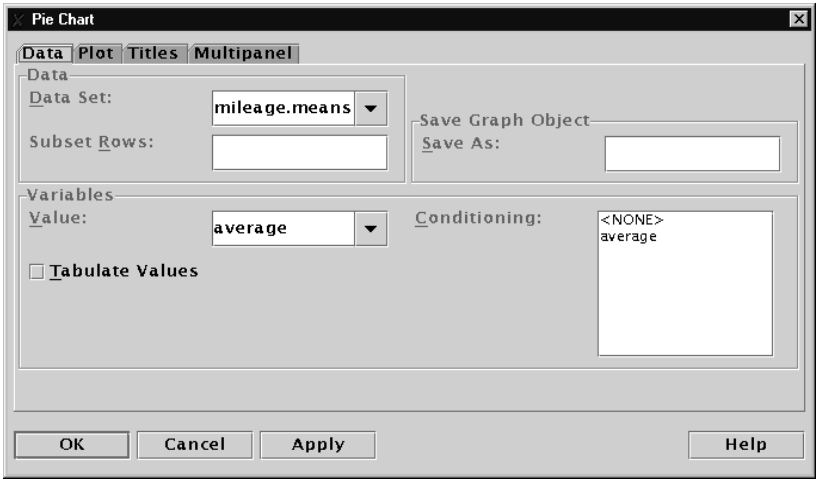


Figure 6.25: *The **Pie Chart** dialog.*

Example 1

In the section Bar Charts on page 166, we used bar charts to graphically display the `mileage.means` data set. In this example, we create a pie chart of these data.

1. If you have not done so already, create the `mileage.means` data set with the instructions given on page 167.
2. Open the **Pie Chart** dialog.
3. Type `mileage.means` in the **Data Set** field.
4. Select `average` as the **Value**.
5. Deselect the **Tabulate Values** option.
6. Click **Apply** to leave the dialog open.

By default, S-PLUS includes a legend to match the pie wedges with their labels. If you would like to include labels on the slices instead, click on the **Plot** tab in the open **Pie Chart** dialog. Deselect the **Include Legend** option and check the boxes for **Include Slice Labels** and **Rotate Labels**. Click **OK**, and a new **Graph** window appears displaying the changes you made. The result is similar to Figure 6.26.

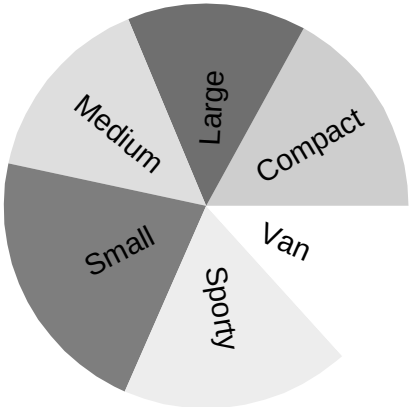


Figure 6.26: *Pie chart of the* `mileage.means` *data.*

Because the average mileage of each type of car cannot be easily interpreted as a fraction of the total mileage, Figure 6.26 does not convey the information in `mileage.means` very well. We can see that small cars get slightly better mileage on average, since the corresponding pie wedge is the largest in the chart. Other than that, the size of the pie wedges simply imply that the mileage of the cars are relatively close in value when compared to the sum total. To refine these conclusions, we would need to view a bar chart or a dot plot of the data.

Example 2

In this example, we tabulate the number of cars in the `fuel.frame` data set for each level of the `Type` factor variable.

1. Open the **Pie Chart** dialog.
2. Type `fuel.frame` in the **Data Set** field. Select `Type` as the **Value**.
3. Verify that the **Tabulate Values** option is checked and click **OK**.

A **Graph** window appears that displays a pie chart of the tabulated values in the `fuel.frame` data set. A pie chart makes more visual sense in this example than it did in the previous example, because each level of `Type` can be viewed as a fraction of the total number of observations in `fuel.frame`.

VISUALIZING TWO-DIMENSIONAL DATA

Two-dimensional data are often called *bivariate* data, and the individual, one-dimensional components of the data are referred to as *variables*. Two-dimensional plots help you quickly grasp the nature of the relationship between the two variables that constitute bivariate data. For example, you might want to know whether the relationship is linear or nonlinear, if the variables are highly correlated, if there any outliers or distinct clusters, etc. In this section, we examine a number of basic plot types useful for exploring a two-dimensional data object.

- **Box Plot**: a graphical representation showing the center and spread of a distribution, as well as any outlying data points.
- **Strip Plot**: a one-dimensional scatter plot.
- **QQ Plot**: a powerful tool for comparing the distributions of two sets of data.

When you couple two-dimensional plots of bivariate data with one-dimensional visualizations of each variable's distribution, you gain a thorough understanding of your data.

Box Plots

A *box plot*, or box and whisker plot, is a clever graphical representation showing the center and spread of a distribution. A box is drawn that represents the bulk of the data, and a line or a symbol is placed in the box at the median value. The width of the box is equal to the *interquartile range*, or IQR, which is the difference between the third and first quartiles of the data. The IQR indicates the spread of the distribution for the data. Whiskers extend from the edges of the box to either the extreme values of the data, or to a distance of $1.5 \times \text{IQR}$ from the median, whichever is less. Data points that fall outside of the whiskers may be outliers, and are therefore indicated by additional lines or symbols.

By default, S-PLUS generates horizontal box plots from the menu options. If you require vertical box plots, you should use the command line function `boxplot`.

Creating a box plot

From the main menu, choose **Graph ▶ Two Variables ▶ Box Plot**. The **Box Plot** dialog opens, as shown in Figure 6.27.

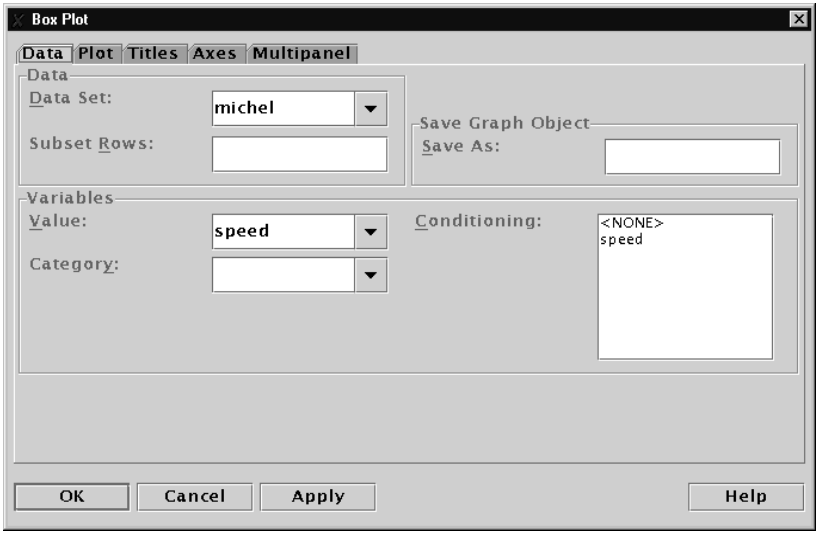


Figure 6.27: *The **Box Plot** dialog.*

Example 1

In the section Density Plots on page 158, we created a probability density estimate for the `michel` data. In this example, we view a box plot of the data.

1. If you have not done so already, create the `michel` data set with the instructions given on page 160.
2. Open the **Box Plot** dialog.
3. Type `michel` in the **Data Set** field.
4. Select `speed` as the **Value** and leave the **Category** field blank.
5. Click **Apply** to leave the dialog open.

The result is shown in Figure 6.28.

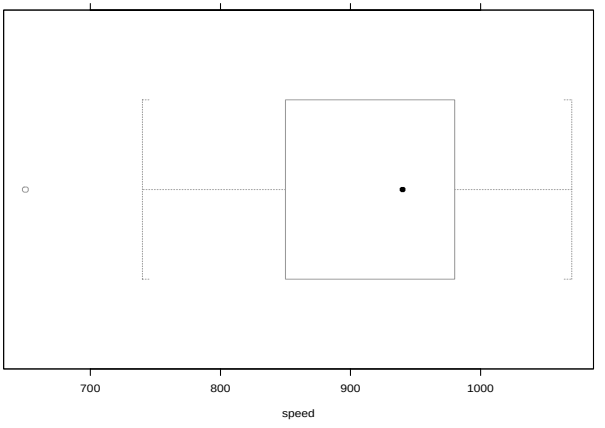


Figure 6.28: *Box plot of the Michelson data.*

The symbol used to indicate the median in each of the boxes is a solid circle by default. To change the symbol, click on the **Plot** tab in the open **Box Plot** dialog. Choose a new symbol from the **Select Symbol** list, and click **Apply** to see the changes. When you are finished experimenting, click **OK** to close the dialog.

Example 2

The `lottery.payoff`, `lottery2.payoff`, and `lottery3.payoff` vectors contain the payoffs for the winning 3-digit numbers in the New Jersey State Pick-It lottery. The `lottery.payoff` object contains 254 values, corresponding to the drawings from May 22, 1975 to March 16, 1976. The `lottery2.payoff` object contains 254 values corresponding to drawings from the 1976-1977 lottery, and `lottery3.payoff` contains 252 values corresponding to the 1980-1981 lottery. In this example, we examine the distributions of these data using box plots.

To create a data frame of the lottery payoff vectors that is suitable for the **Box Plot** dialog, we can use the `make.groups` function:

```
> lottery.payoffs <- make.groups(
+ "1975" = lottery.payoff,
+ "1977" = lottery2.payoff,
+ "1981" = lottery3.payoff)
```

```
> lottery.payoffs

    data which
 1 190.0  1975
 2 120.5  1975
 3 285.5  1975
 4 184.0  1975
 5 384.5  1975
 6 324.5  1975
 7 114.0  1975
 8 506.5  1975
 9 290.0  1975
10 869.5  1975
11 668.5  1975
12  83.0  1975
13 . . .
```

The `data` column is a numeric variable containing the payoff values from each of the three vectors. The `which` column is a factor variable with three levels, corresponding to the chosen names `"1975"`, `"1977"`, and `"1981"`. Thus, `lottery.payoff` appears at the beginning of the data frame, `lottery2.payoff` is in the middle, and `lottery3.payoff` is at the end of the data set.

Once you have generated the `lottery.payoffs` data, create a box plot as follows:

1. Open the **Box Plot** dialog.
2. Type `lottery.payoffs` in the **Data Set** field.
3. Select `data` as the **Value**.
4. Select `which` as the **Category**.
5. Click **OK**.

The result is displayed in Figure 6.29.

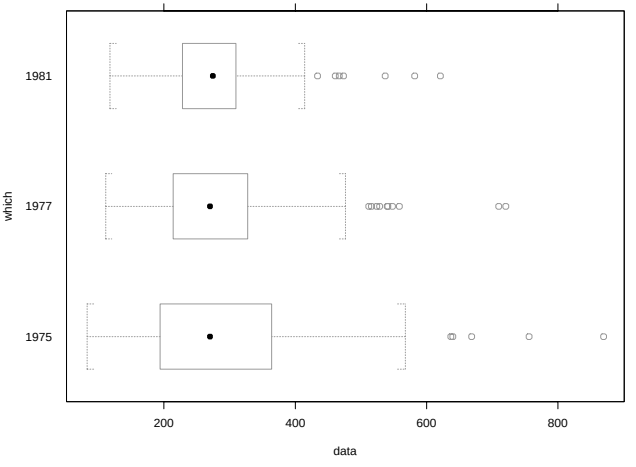


Figure 6.29: *Box plots of the* `lottery.payoffs` *data.*

Strip Plots

A *strip plot* can be thought of as a one-dimensional scatter plot. Strip plots are similar to box plots in overall layout, but they display all of the individual data points instead of the box plot summary.

Creating a strip plot

From the main menu, choose **Graph ▶ Two Variables ▶ Strip Plot**. The **Strip Plot** dialog opens, as shown in Figure 6.30.

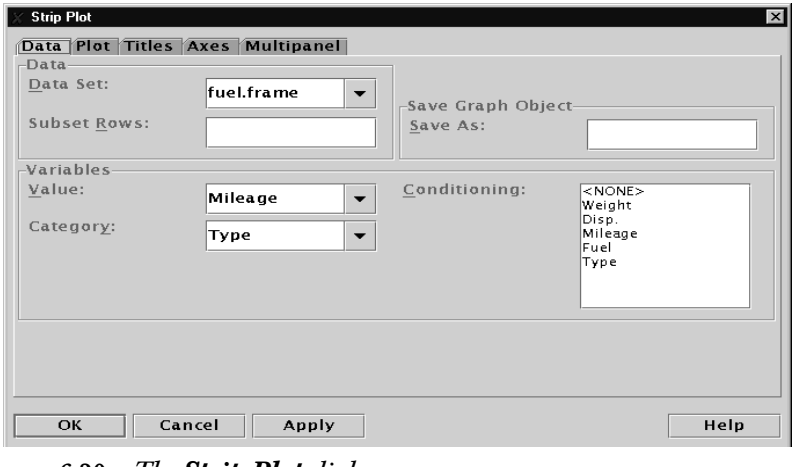


Figure 6.30: *The* ***Strip Plot*** *dialog.*

Example

In this example, we graphically analyze the mileage for each of the six types of cars in the `fuel.frame` data.

1. Open the **Strip Plot** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. Select `Mileage` as the **Value** and `Type` as the **Category**.
4. Click on the **Titles** tab and select **<NONE>** for the **Y Axis Label**.
5. Click **Apply** to leave the dialog open.

At first glance, there appears to be very few points in the strip plot. This is because points with the same x coordinate overlap each other in the horizontal strips. You can distinguish points very near to each other by adding random vertical noise to the points' coordinates. This alleviates some of the overlap in a strip plot's symbols. To do this, click on the **Plot** tab in the open **Strip Plot** dialog and check the **Jitter Symbols Vertically** option. Click **OK** to close the dialog and see the updated graph. The result is shown in Figure 6.31.

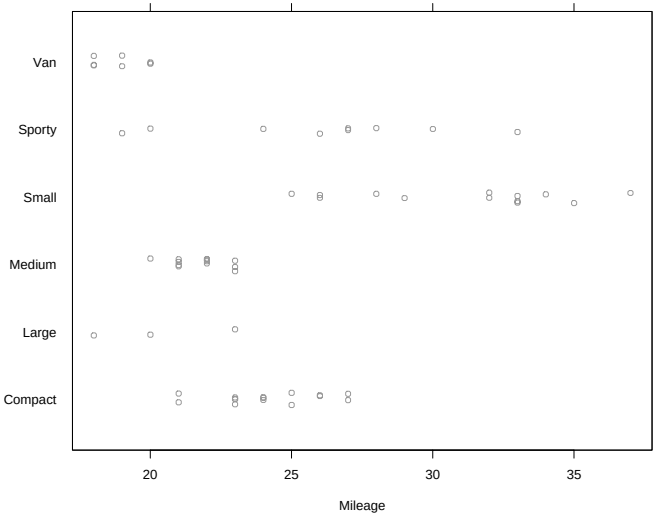


Figure 6.31: *Strip plot of mileage in the* `fuel.frame` *data set.*

QQ Plots

In the section Visualizing One-Dimensional Data, we introduced the quantile-quantile plot, or *qqplot*, as an extremely powerful tool for determining a good approximation to a data set's distribution. In a one-dimensional qqplot, the ordered data are graphed against quantiles of a known theoretical distribution. If the data points are drawn from the theoretical distribution, the resulting plot is close to a straight line in shape.

We can also use qqplots with two-dimensional data to compare the distributions of the variables. In this case, the ordered values of the variables are plotted against each other. If the variables have the same distribution shape, the points in the qqplot cluster along a straight line. The **QQ Plot** dialog creates a qqplot for the two groups in a binary variable. It expects a numeric variable and a factor variable with exactly two levels; the values of the numeric variable corresponding to each level are then plotted against each other.

Creating a QQ plot

From the main menu, choose **Graph ▶ Two Variables ▶ QQ Plot**. The **QQ Plot** dialog opens, as shown in Figure 6.32.

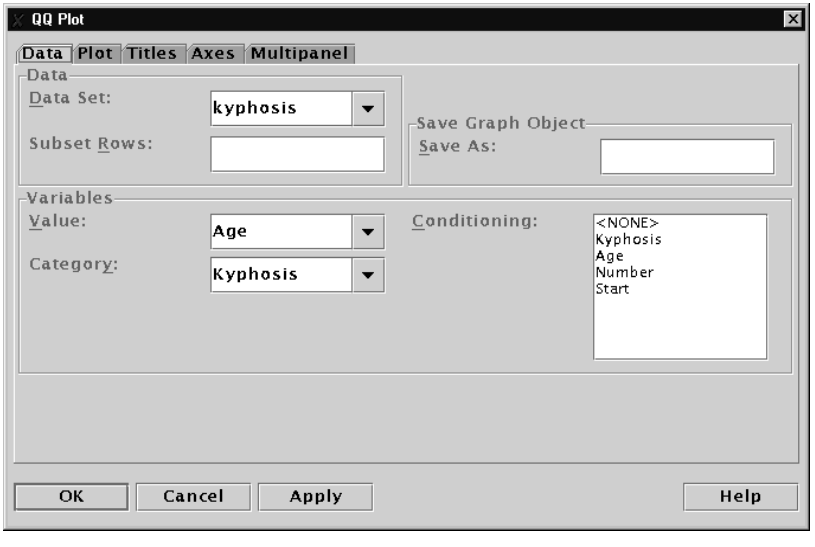


Figure 6.32: *The **QQ Plot** dialog.*

Example

The `kyphosis` data set has 81 rows representing data on 81 children who have had corrective spinal surgery. The outcome `Kyphosis` is a binary variable, and the other three columns `Age`, `Number`, and `Start`, are numeric. Kyphosis is a post-operative deformity which is present in some children receiving spinal surgery. We are interested in examining whether the child's age, the number of vertebrae operated on, or the starting vertebra influence the likelihood of the child having a deformity. As an exploratory tool, we test whether the distributions of `Age`, `Number`, and `Start` are the same for the children with and without kyphosis. To do this, we create qqplots for each of the variables.

1. Open the **QQ Plot** dialog.
2. Type `kyphosis` in the **Data Set** field.
3. Select `Kyphosis` as the **Category**.
4. Select `Age` as the **Value**. Click on the **Titles** tab and type `Age` for the **Main Title**. Click **Apply**.
5. Click on the **Data** tab and select `Number` as the **Value**. Change the **Main Title** to `Number` and click **Apply**.
6. Click on the **Data** tab and select `Start` as the **Value**. Change the **Main Title** to `Start` and click **OK**.

By default, S-PLUS includes a reference line in qqplots. To omit the line from a graph, deselect the **Include Reference Line** option in the **Plot** page of the dialog.

The three qqplots appear in separate **Graph** windows. The only variable that clusters near the straight line drawn in the qqplots is `Age`, as shown in Figure 6.33. This suggests that the `Age` values corresponding to the two levels in `Kyphosis` come from roughly the same distribution. In other words, the children with and without kyphosis do not differ significantly in the distribution of their ages. On the other hand, the children do differ significantly in the distributions of how many vertebrae were involved in the operation, as well as which vertebra was the starting vertebra.

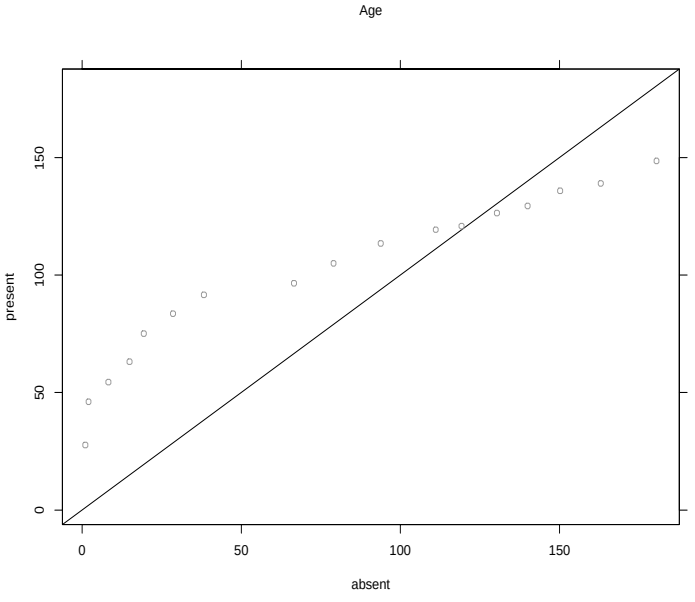


Figure 6.33: *Normal qqplot of* `Age`*, for the two groups in the binary* `Kyphosis` *variable.*

VISUALIZING THREE-DIMENSIONAL DATA

Three-dimensional data have three columns, or *variables*, of univariate data, and the relationships between variables form a surface in 3D space. Because the depth cues in three-dimensional plots are sometimes insufficient to convey all of the information, special considerations must be made when visualizing three-dimensional data. Instead of viewing the surface alone, we can analyze projections, slices, or rotations of the surface. In this section, we examine a number of basic plot types useful for exploring a three-dimensional data object.

- **Contour Plot**: uses contour lines to represent heights of three-dimensional data in a flat, two-dimensional plane.
- **Level Plot**: uses colors to represent heights of three-dimensional data in a flat, two-dimensional plane. Level plots and contour plots are essentially identical, but they have defaults that allow you to view a particular surface differently.
- **Surface Plot**: approximates the shape of a data set in three dimensions.
- **Cloud Plot**: displays a three-dimensional scatter plot of points.

Contour Plots

A *contour plot* is a representation of three-dimensional data in a flat, two-dimensional plane. Each contour line represents a height in the z direction from the corresponding three-dimensional surface. Contour plots are often used to display data collected on a regularly-spaced grid; if gridded data is not available, interpolation is used to fit and plot contours.

Creating a contour plot

From the main menu, choose **Graph ► Three Variables ► Contour Plot**. The **Contour Plot** dialog opens, as shown in Figure 6.34.

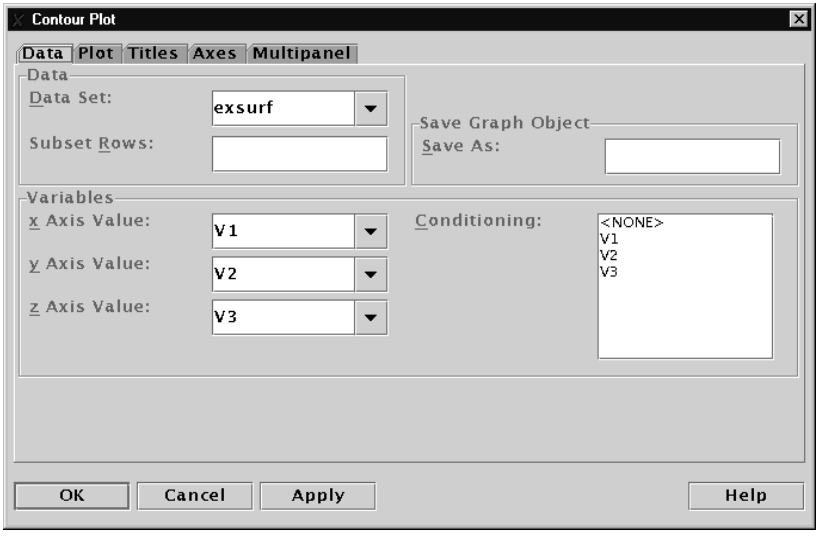


Figure 6.34: *The **Contour Plot** dialog.*

Example

The `exsurf` data set has 1271 rows and 3 columns: `V1`, `V2`, and `V3`. It is an example data set that is useful for demonstrating the functionality of three-dimensional plots over a regular grid. In this example, we use contour plots to explore the shape of the `exsurf` data.

1. Open the **Contour Plot** dialog.
2. Type `exsurf` in the **Data Set** field.
3. Select `V1` as the **x Axis Value**, `V2` as the **y Axis Value**, and `V3` as the **z Axis Value**.
4. Click **Apply** to leave the dialog open.

The result is shown in Figure 6.35.

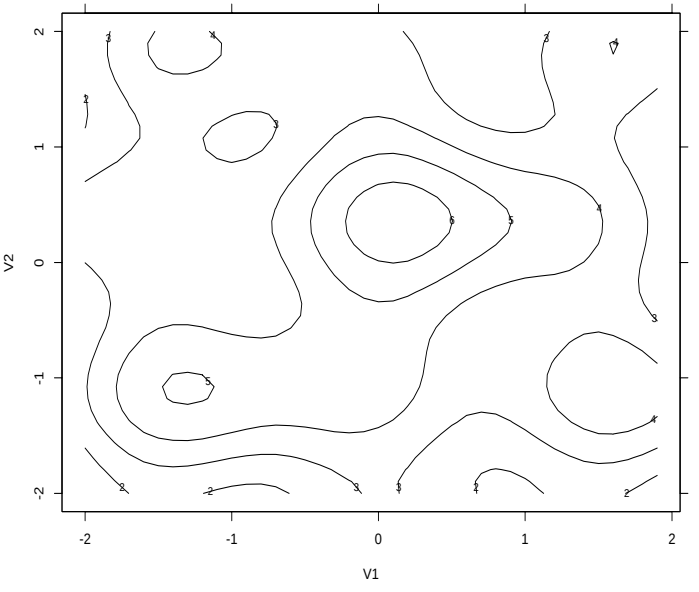


Figure 6.35: *Contour plot of the* `exsurf` *data.*

By default, S-PLUS uses 7 slices through the three-dimensional surface to produce the lines in a contour plot. If you want to increase or decrease the number of contour lines, click on **Plot** tab in the open **Contour Plot** dialog and enter a new value for the **Number of Cuts**. The **Use Pretty Contour Levels** option determines whether the contour lines are chosen at rounded z values, which allows them to be labelled clearly. When you are finished experimenting, click **OK** to close the dialog.

Level Plots

A *level plot* is essentially identical to a contour plot, but it has default options that allow you to view a particular surface differently. Like contour plots, level plots are representations of three-dimensional data in flat, two-dimensional planes. Instead of using contour lines to indicate heights in the z direction, however, level plots use colors. Specifically, level plots include color fills and legends by default, and they do not include contour lines or labels.

Creating a level plot

From the main menu, choose **Graph ▶ Three Variables ▶ Level Plot**. The **Level Plot** dialog opens, as shown in Figure 6.36.

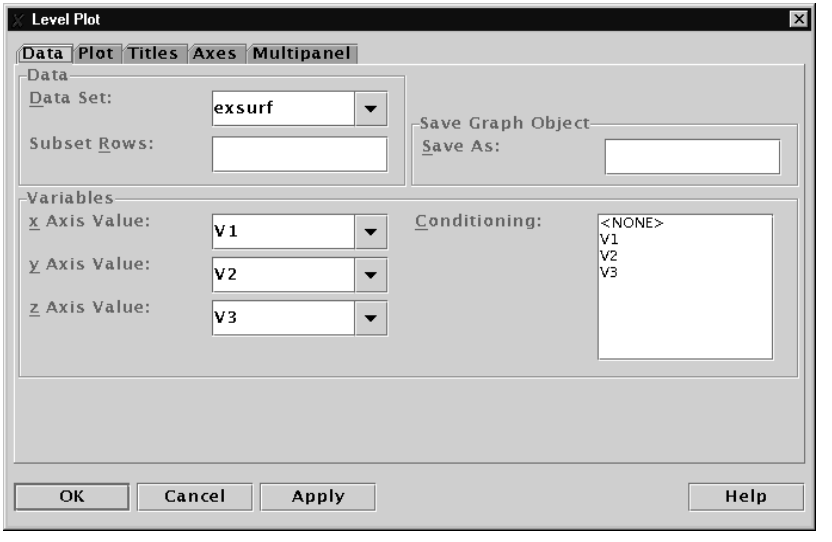


Figure 6.36: *The **Level Plot** dialog.*

Example

In this example, we use level plots to explore the shape of the `exsurf` data set.

1. Open the **Level Plot** dialog.
2. Type `exsurf` in the **Data Set** field.
3. Select `V1` as the **x Axis Value**, `V2` as the **y Axis Value**, and `V3` as the **z Axis Value**.
4. Click **OK**.

A **Graph** window appears that displays the level plot and its corresponding legend.

Surface Plots

A *surface plot* is an approximation to the shape of a three-dimensional data set. Surface plots are used to display data collected on a regularly-spaced grid; if gridded data is not available, interpolation is used to fit and plot the surface.

Creating a surface plot

From the main menu, choose **Graph ▶ Three Variables ▶ Surface Plot**. The **Surface Plot** dialog opens, as shown in Figure 6.37.

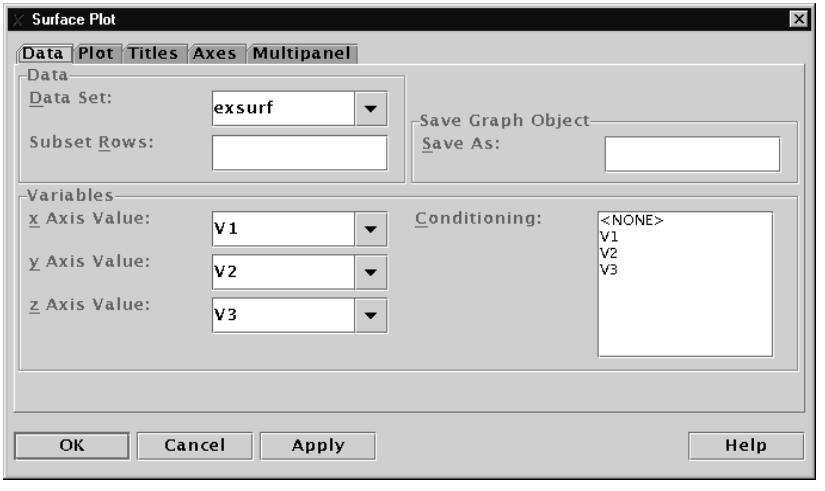


Figure 6.37: *The **Surface Plot** dialog.*

Example

In this example, we create a surface plot of the `exsurf` data set.

1. Open the **Surface Plot** dialog.
2. Type `exsurf` in the **Data Set** field.
3. Select `V1` as the **x Axis Value**, `V2` as the **y Axis Value**, and `V3` as the **z Axis Value**.
4. Click **Apply** to leave the dialog open.

The result is shown in Figure 6.38.

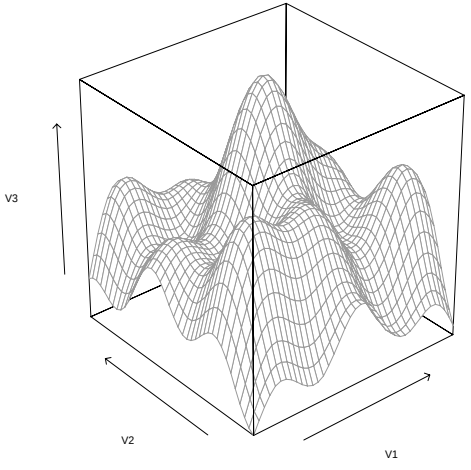


Figure 6.38: *Surface plot of the* `exsurf` *data.*

The arrows along the axes in Figure 6.38 indicate the direction of increasing values for each of the variables. To include tick marks instead of arrows, click on the **Axes** tab in the open **Surface Plot** dialog and check the **Include Tick Marks and Labels** box.

By default, S-PLUS rotates a surface plot 40 degrees about the z axis and -60 degrees about the x axis before displaying it. To change this setting, enter new values in the **Rotation** fields; rotating each axis 0 degrees results in a view from the top of the surface, looking down in the x-y plane. The **Distance Factor** controls the distance from the surface to the viewer. A distance factor of 0 implies the viewer is right at the object, and a factor of 1 implies the viewer is infinitely far away. The **Zoom Factor** controls the overall scaling for the drawn surface. Zoom values larger than 1 enlarge the object, and values less than 1 compress the object.

If you would like to create a surface plot with colors, click on the **Plot** tab in the open **Surface Plot** dialog and check the **Include Fills** box. Click **OK** to close the dialog, and a new **Graph** window appears that displays the changes you made.

Cloud Plots

A *cloud plot* is a three-dimensional scatter plot of points. Typically, a static 3D scatter plot is not effective because the depth cues of single points are insufficient to give a strong 3D effect. On some occasions, however, cloud plots can be useful for discovering simple characteristics about the three variables.

Creating a cloud plot

From the main menu, choose **Graph ► Three Variables ► Cloud Plot**. The **Cloud Plot** dialog opens, as shown in Figure 6.39.

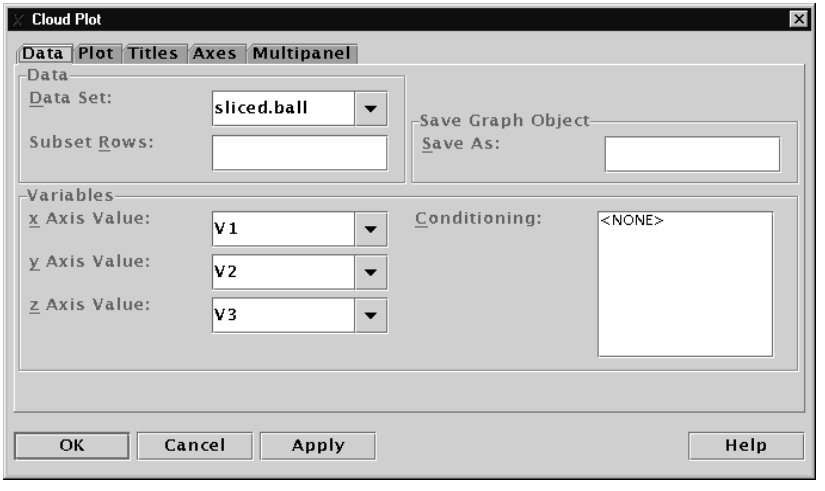


Figure 6.39: *The **Cloud Plot** dialog.*

Example

The `sliced.ball` data set contains three variables that comprise a set of points uniformly distributed in a three-dimensional sphere, except that a central slice of the points has been removed. The removed slice is oriented so that all two-dimensional projections of the data appear to be uniformly distributed over a disk. In addition, the slice is not visible in the initial three-dimensional view. In this example, we discover the location of the slice by rotating a cloud plot.

1. Open the **Cloud Plot** dialog.
2. Type `sliced.ball` in the **Data Set** field.
3. Select `V1` as the **x Axis Value**, `V2` as the **y Axis Value**, and `V3` as the **z Axis Value**. Click **Apply** to leave the dialog open.

Note that the removed slice of data points is not visible in the initial graph. To rotate the scatter plot, click on the **Axes** tab in the open **Cloud Plot** dialog. The options in the **Axes** tab are identical to those in the **Surface Plot** dialog. Experiment with different **Rotation** values, clicking **Apply** each time you enter a new set of numbers. Each time you click **Apply**, a new **Graph** window appears displaying the rotated view of the surface. In particular, the values of -42, 0, and 40 clearly show the missing slice of data points, as displayed in Figure 6.40. When you are finished experimenting, click **OK** to close the dialog.

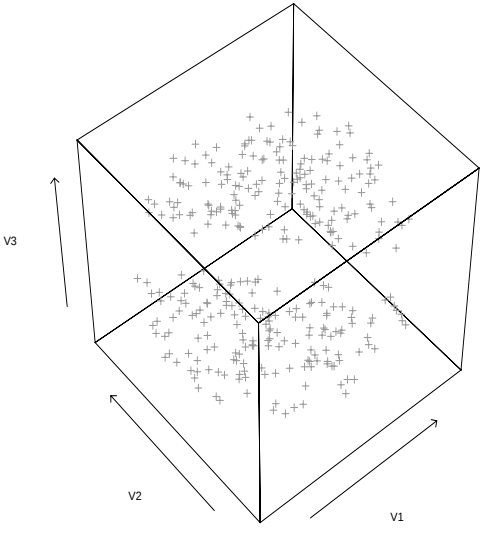


Figure 6.40: *Cloud plot of the* `sliced.ball` *data set, showing the missing slice of data points.*

VISUALIZING MULTIDIMENSIONAL DATA

In the previous sections, we discussed visual tools for simple one-, two-, and three-dimensional data sets. With lower-dimensional data, all of the basic information in the data may be easily viewed in a single set of plots. Different plots provide different types of information, but deciding which plots to use is fairly straightforward.

With multidimensional data, however, visualization is more involved. In addition to univariate and bivariate relationships, variables may have interactions such that the relationship between any two variables changes depending on the remaining variables. Standard one- and two-variable plots do not allow us to look at interactions between multiple variables, and must therefore be complemented with techniques specifically designed for multidimensional data. In this section, we discuss both standard and novel visualization tools for multidimensional data.

- **Scatterplot Matrix**: displays an array of pairwise scatter plots illustrating the relationship between any pair of variables.
- **Parallel Plot**: displays the variables in a data set as horizontal panels, and connects the values for a particular observation with a set of line segments.

Two additional techniques for visualizing multidimensional data are *grouping variables* and *multipanel conditioning*. We briefly discussed both of these tools in the section Scatter Plots, and we intersperse more detailed examples below. The conditioning options that we discuss are not specific to scatter plots, but are available in most dialogs under the **Graph** menu. You can therefore use the options to create multiple histograms, box plots, etc., conditioned on the value of a particular variable in your data set.

Scatterplot Matrices

A *scatterplot matrix* is a powerful graphical tool that enables you to quickly visualize multidimensional data. It is an array of pairwise scatter plots illustrating the relationship between any pair of variables in a multivariate data set. Often, when faced with the task of analyzing data, the first step is to become familiar with the data. Generating a scatterplot matrix greatly facilitates this process.

Creating a scatterplot matrix

From the main menu, choose **Graph ▶ Multiple Variables ▶ Scatterplot Matrix**. The **Scatterplot Matrix** dialog opens, as shown in Figure 6.41.

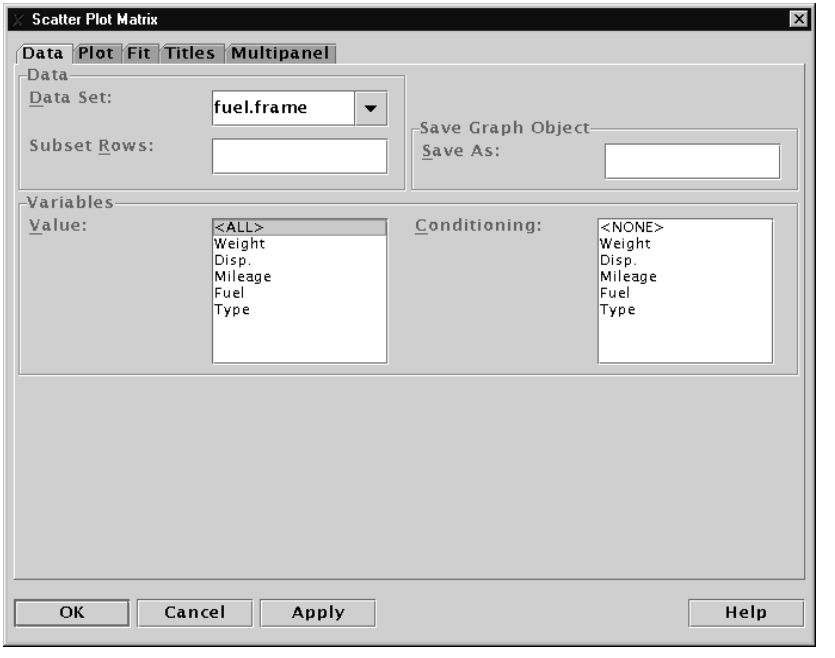


Figure 6.41: *The **Scatterplot Matrix** dialog.*

Example

In this example, we create a scatterplot matrix of the `fuel.frame` data.

1. Open the **Scatterplot Matrix** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. Select **<ALL>** in the **Variables** box to create a 5×5 scatterplot matrix that includes all variables.
4. Click **Apply** to leave the dialog open.

The result is shown in Figure 6.42.

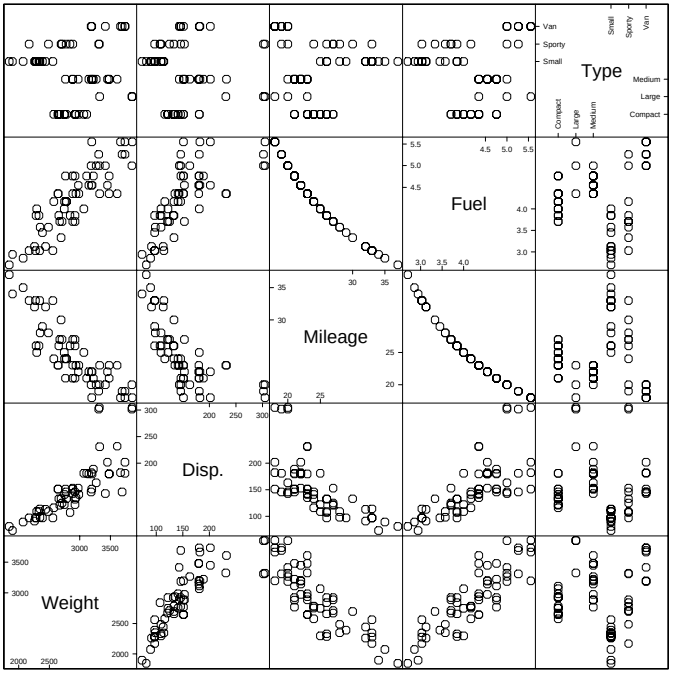


Figure 6.42: *Scatterplot matrix of the* `fuel.frame` *data. A number of strong relationships appears.*

From the figure, you can immediately see a number of strong linear relationships. For example, the weight of a car and its fuel consumption have a positive linear relationship: as `Weight` increases, so does `Fuel`. Note that the factor variable `Type` has been converted to a numeric variable and plotted. The six levels of `Type` (`Compact`, `Large`, `Medium`, `Small`, `Sporty`, and `Van`) simply take the values 1 through 6 in this conversion.

The **Scatterplot Matrix** dialog contains the same options as the **Scatter Plot** dialog for grouping variables, fitting lines, and smoothing. Thus, you can add curve fits or distinguish the levels of a grouping variable in each of the panels of a scatterplot matrix. For example, to add least squares line fits to each of the plots in Figure 6.42, click on the **Fit** tab in the open S**catterplot Matrix** dialog. Select **Least Squares** as the **Regression Type** and click **OK**. As an

additional example, the following steps create a matrix of the four numeric variables in `fuel.frame`, distinguishing the different levels of `Type` in each scatter plot:

1. Open the **Scatterplot Matrix** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. CTRL-click to highlight `Weight`, `Disp.`, `Mileage`, and `Fuel` in the **Variables** box.
4. Click on the **Plot** tab. Select `Type` in the **Group Variable** list, and check the boxes for **Vary Symbol Style** and **Include Legend**.
5. Click **OK**.

A new **Graph** window appears displaying the scatterplot matrix.

Parallel Plots

A *parallel coordinates plot* displays the variables in a data set as horizontal panels, and connects the values for a particular observation with a set of line segments. These kinds of plots show the relative positions of observation values as coordinates on parallel horizontal panels.

Creating a parallel plot

From the main menu, choose **Graph ▶ Multiple Variables ▶ Parallel Plot**. The **Parallel Plot** dialog opens, as shown in Figure 6.43.

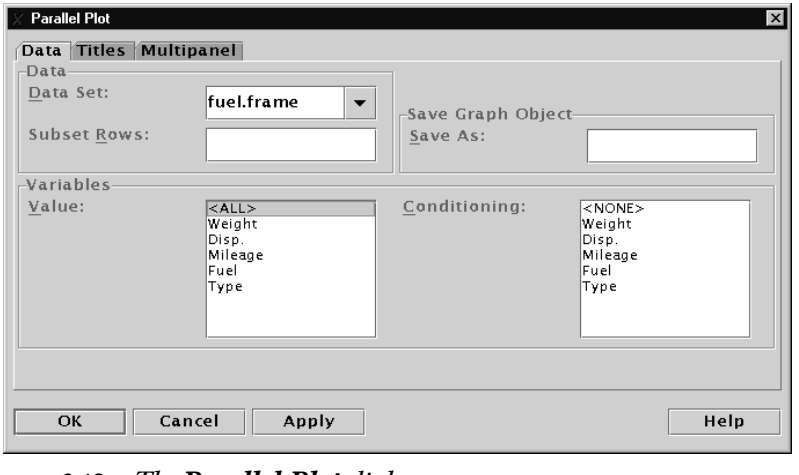


Figure 6.43: *The **Parallel Plot** dialog.*

Example

In this example, we create a parallel coordinates plot of the `fuel.frame` data.

1. Open the **Parallel Plot** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. Select **<ALL>** in the **Variables** box to create a 5-panel plot that includes all variables.
4. Click **OK**.

The result is shown in Figure 6.44.

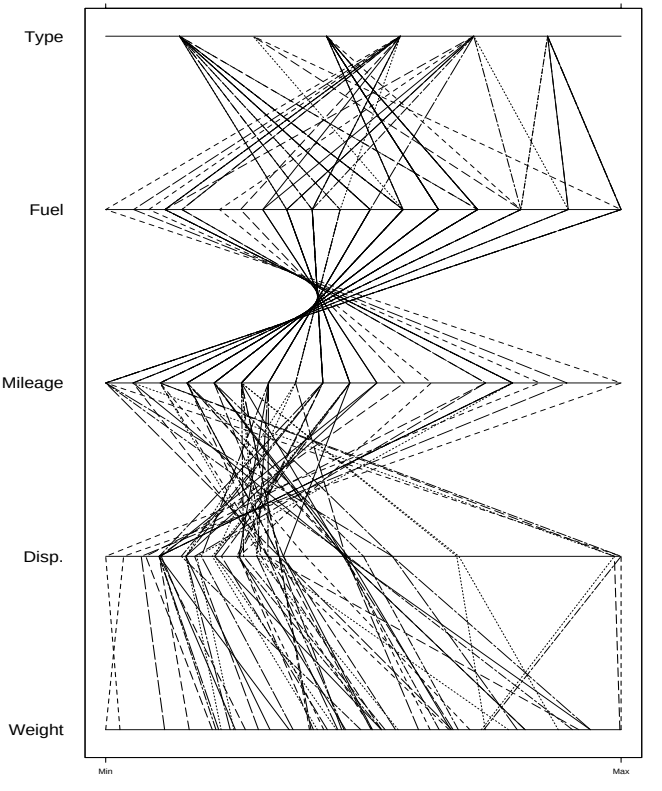


Figure 6.44: *Parallel coordinates plot of the* `fuel.frame` *data set.*

Multipanel Trellis Graphics

Trellis graphics allow you to view relationships between different variables in your data set through conditioning. Suppose you have a data set based on multiple variables, and you want to see how plots of two variables change in relation to a third "conditioning" variable. With Trellis graphics, you can view your data in a series of panels, where each panel contains a subset of the original data divided into intervals of the conditioning variable. When a conditioning variable is categorical, S-PLUS generates plots for each level. When a conditioning variable is numeric, conditioning is automatically carried out on the sorted unique values; each plot represents either an equal number of observations or an equal range of values.

A wide variety of graphs can be conditioned using Trellis graphics, and many of the dialogs under the **Graph** menu include Trellis display options. In the section Scatter Plots, we illustrate how conditioning can be used with scatter plots to reveal relationships in multivariate data. In this section, we present another detailed example that shows the functionality of Trellis graphics.

Example

The `barley` data set contains observations from a 1930s agricultural field trial that studied barley crops. At six sites in Minnesota, ten varieties of barley were grown for each of two years, 1931 and 1932. The data are the yields for all combinations of site, variety, and year, so there are a total of $6 \times 10 \times 2 = 120$ observations. The data first appeared in a 1934 report published by the experimenters, and has been analyzed and re-analyzed ever since. R.A. Fisher presented the data for five of the sites in his classic book, *The Design of Experiments* (1971). Publication in the book made the data famous; many other statisticians subsequently analyzed the data, usually to illustrate a new statistical method.

In the early 1990s, Bill Cleveland of AT&T (now Lucent Technologies) analyzed the `barley` data using Trellis graphics. The results were quite surprising, and the basis of Cleveland's analysis is repeated here for illustrative purposes. For historical details about the barley experiment, see the Cleveland (1993) reference.

Exploratory data analysis

We are interested in exploring how barley yield varies based on combinations of the `variety`, `year`, and `site` variables. Trellis graphics are particularly useful for displaying effects and interactions between variables. We create a scatter plot of `yield` and `variety` conditioned on `site`, and vary the plotting symbol by `year`. Because `site` is a factor variable with six levels, our Trellis graph will have six panels labeled with the names of the sites. In addition, `year` is a factor variable with two levels, so each panel in our Trellis graph will include two different plotting symbols.

1. Select **Graph ► Scatter Plot** to open the **Scatter Plot** dialog.
2. Type barley in the **Data Set** field. Select `yield` as the **x Axis Value** and `variety` as the **y Axis Value**. Highlight `site` in the **Conditioning** box.
3. Click on the **Plot** tab and select `year` as the **Group Variable**. Check the boxes for **Vary Symbol Style** and **Include Legend**.
4. Click on the **Titles** tab. Type **Bushels/Acre** for the **X Axis Label** and **Variety of Barley** for the **Y Axis Label**.
5. Click on the **Axes** tab and select **Horizontal** for the **Tick Marks Label Orientation**. This option places horizontal tick labels on both the *x* and *y* axes. By default, labels are parallel to the axes, so that *x* axis tick labels are horizontal and *y* axis labels are vertical.
6. Click **Apply** to leave the dialog open.

The resulting graph is shown in Figure 6.45.

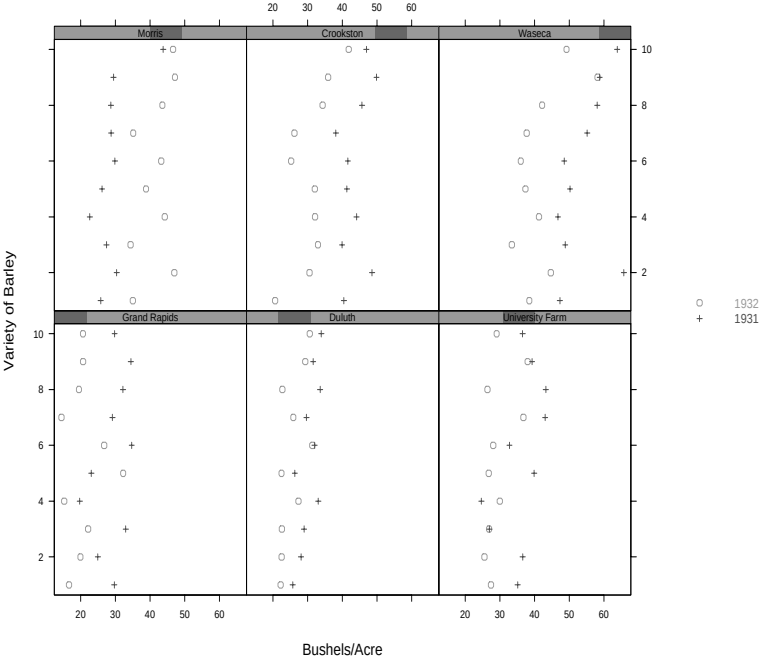


Figure 6.45: *Unformatted Trellis plot of barley yields for 1931 and 1932.*

To simplify the comparison of barley yields across sites, we make two changes to the layout of the panels in Figure 6.45:

1. First, we stack the six panels in one column. To do this, click on the **Multipanel Conditioning** tab in the open **Scatter Plot** dialog. Type 1 for the **# of Columns** and 6 for the **# of Rows**.

2. Next, we set the aspect ratio of each panel to 0.5. To do this, click on the **Axes** tab in the open **Scatter Plot** dialog. Set the **Aspect Ratio** to be a **Specified Value** and type 0.5 as the **Ratio Value**.

Click **OK** to close the dialog, and a new **Graph** window appears that displays the updated set of plots. The final Trellis graphic looks similar to the one shown in Figure 6.46.

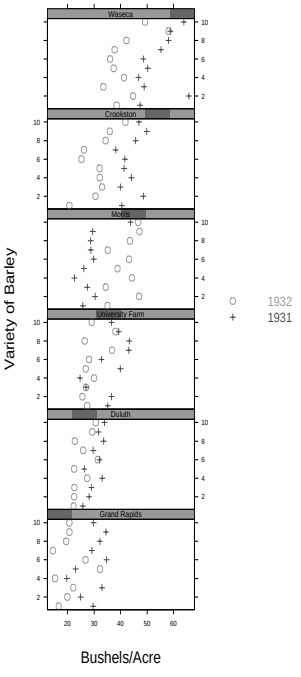


Figure 6.46: *Formatted Trellis plot of barley yields for 1931 and 1932.*

Examine Figure 6.46 to find a discrepancy in the `barley` data. It appears in the `Morris` panel: for all other sites, 1931 has significantly higher overall yields than 1932, but the reverse is true at the Morris site. More importantly, the amount by which the 1932 yield exceeds the 1931 yield at Morris is similar to the amounts by which 1931 exceeds 1932 at the other five sites. Either an extraordinary natural event (such as disease or a local weather anomaly) produced a strange coincidence, or the years for the Morris data were inadvertently reversed. More Trellis graphics, statistical modeling of the data, and some background checks on the experiment led to the conclusion that the data are in error. But it was a Trellis graphic like the one in Figure 6.46 that originally led Cleveland to this conclusion.

TIME SERIES

Time series are multivariate data sets that are associated with a set of ordered positions, where the positions are an important feature of the values and their analysis. These data can arise in many contexts. For example, in the financial marketplace, trading tickers record the price and quantity of each trade at particular times throughout the day. Such data can be analyzed to assist in making market predictions. This section discusses three plots that are helpful in visualizing time series data.

- **Line Plots**: successive values of the data are connected by straight lines.
- **High-Low Plots**: vertical lines are used to indicate the daily, monthly, or yearly extreme values in a time series, and hatch marks are drawn on the lines to represent the opening and closing values. This type of plot is most often used to display financial data.
- **Stacked Bar Plots**: multiple *y* values determine segment heights in a bar chart.

Note that the dialogs for these time series plots recognize objects of class `"timeSeries"` only, and do not accept data frames, matrices, or vectors. For this reason, we periodically drop to the **Commands** window in this section to create objects that are accepted by the menu options.

Line Plots

With time series data, it is often useful to view a *line plot*, where the successive values of the data are connected by straight lines. By using straight line segments to connect the points, you can see more clearly the overall trend or shape in the ordered data values.

Creating a line plot

From the main menu, choose **Graph ► Time Series ► Line Plot**. The **Time Series Line Plot** dialog opens, as shown in Figure 6.47.

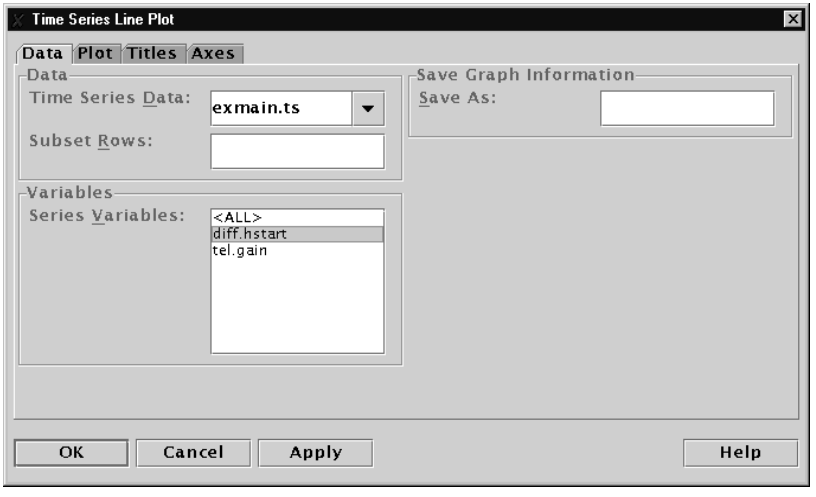


Figure 6.47: *The **Time Series Line Plot** dialog.*

Example

In the section Scatter Plots on page 132, we created the `exmain` data set. The variables in `exmain` are both time series: `tel.gain` and `diff.hstart` contain values recorded once per year on the first of January for the 14 years beginning in 1971. In this example, we use the **Time Series Line Plot** dialog to analyze these variables.

If you have not done so already, create the `exmain` data set with the instructions given on page 134. The `exmain` data is stored in an object of class `"data.frame"`. We must therefore convert it to class `"timeSeries"` before it can be recognized by the dialogs under the **Time Series** menu. To do this, type the following in the **Commands** window:

```
> exmain.ts <- timeSeries(exmain,
+ from = timeCalendar(d=1, m=1, y=1971), by = "years")
```

The `from` and `by` arguments in the call to `timeSeries` define the appropriate units for the time series data.

Exploratory data analysis

To begin our analysis, we create a line plot of `diff.hstart`:

1. Open the **Time Series Line Plot** dialog.
2. Type `exmain.ts` in the **Time Series Data** field.
3. Highlight `diff.hstart` in the **Series Variables** box.
4. Click on the **Titles** tab and type **New Housing Starts** for the **Y Axis Label**.
5. Click **Apply** to leave the dialog open.

The result is shown in Figure 6.48. The fourteen values in `diff.hstart`, representing observations made in the years 1971-1984, are plotted sequentially.

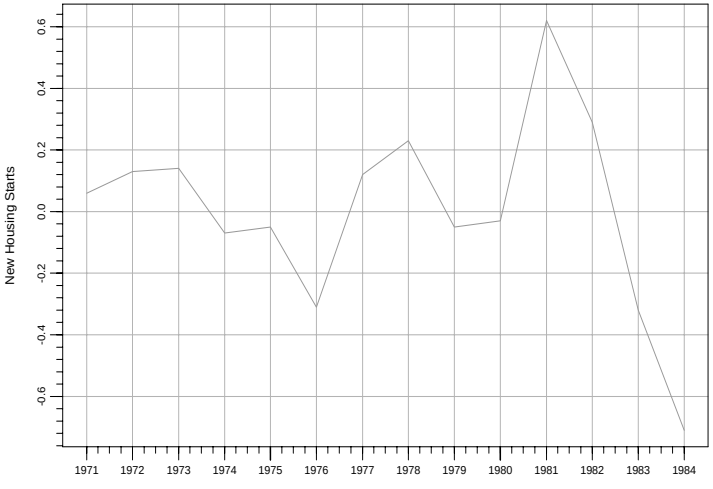


Figure 6.48: *A time series line plot of* `diff.hstart`.

By default, S-PLUS includes a reference grid in time series line plots. To leave the grid out of your graphics, click on the **Axes** tab in the open **Time Series Line Plot** dialog and deselect the **Include Reference Grid** option. To include both points and lines in the graph, click on the **Plot** tab and select **Both Points & Lines** from the **Type** list. When you are finished experimenting, click **OK** to close the dialog.

Now that you have seen the time series behavior of `diff.hstart`, you may be interested in seeing that of `tel.gain` as well. The steps below place line plots of both variables on the same set of axes:

1. Open the **Time Series Line Plot** dialog.
2. Type `exmain.ts` in the **Time Series Data** field.
3. CTRL-click to highlight `diff.hstart` and `tel.gain` in the **Series Variables** box.
4. Click on the **Plot** tab and select **Both Points & Lines** from the **Type** list. Check the boxes for **Vary Line Style** and **Include Legend**.
5. Click on the **Titles** tab and type **The Main Gain Data** as the **Main Title**.
6. Click **OK**.

The result is shown in Figure 6.49.

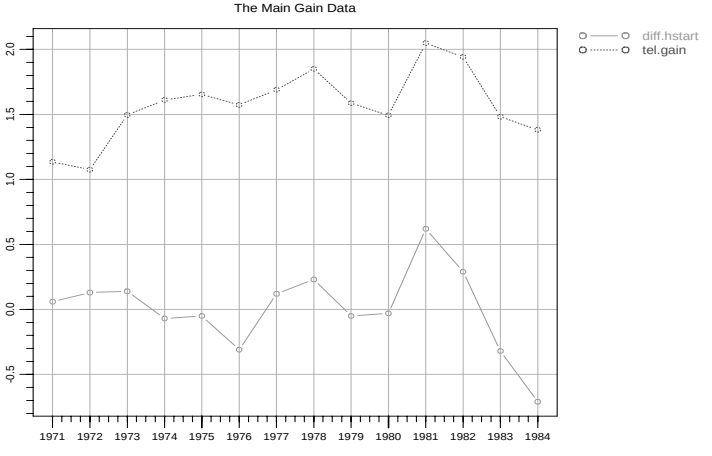


Figure 6.49: *Time series line plots of `tel.gain` and `diff.hstart`.*

Viewing line plots of `tel.gain` and `diff.hstart` is a simple yet powerful complement to viewing scatter plots of these variables alone. Using both plot types gives you a more complete understanding of the data. Earlier in this chapter, we determined that the first two observations in `exmain` were outliers. The time series line plots reveal that the `tel.gain` values during the first two years

were the smallest during the 14-year study. At the same time, the `diff.hstart` values during the first two years were near their overall average for the 14-year time period. Furthermore, notice that except for the first four years, there is a striking correlation pattern between the two variables: whenever one increases, so does the other. In comparison to the final years of the study, it appears that the relative behavior of the two variables is different during the 1971-1974 time period.

High-Low Plots

A *high-low plot* typically displays lines indicating the daily, monthly, or yearly extreme values in a time series. These kinds of plots can also include average, opening, and closing values, and are referred to as *high-low-open-close plots* in these cases. Meaningful high-low plots can thus display from three to five columns of data, and illustrate simultaneously a number of important characteristics about time series data. Because of this, they are most often used to display financial data.

In typical high-low plots, vertical lines are drawn to indicate the range of values in a particular time unit (i.e., day, month, or year). If opening and closing values are included in the plot, they are represented by small horizontal hatch marks on the lines: left-pointing hatch marks indicate opening values and right-pointing marks indicate closing values. One variation on the high-low plot is the *candlestick plot*. Where typical high-low plots display the opening and closing values of a financial series with lines, candlestick plots use filled rectangles. The color of the rectangle indicates whether the difference is positive or negative. In S-PLUS, white rectangles represent positive differences, when closing values are larger than opening values. Blue rectangles indicate negative differences, when opening values are larger than closing values.

Creating a high-low plot

From the main menu, choose **Graph ► Time Series ► High-Low Plot**. The **Time Series High-Low Plot** dialog opens, as shown in Figure 6.50.

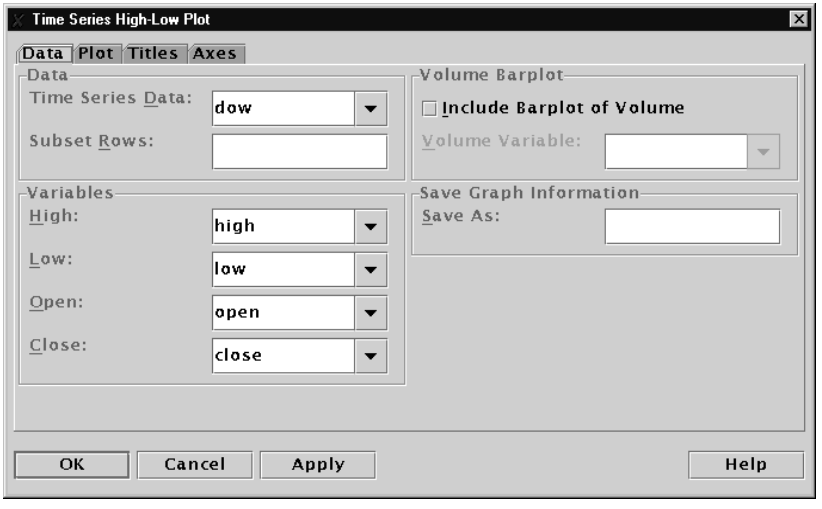


Figure 6.50: *The **Time Series High-Low Plot** dialog.*

Example

The `djia` data set is a multivariate time series taken from the Ohio State University web site. It contains the high, low, opening, and closing prices, as well as the daily trading volume, for the Dow Jones Industrial Average. The data set has the closing price only from 1915 through September 1928, and it contains the high, low, and closing prices from October 1928 through March 9, 1984. The high, low, opening, and closing prices from March 12, 1984 through December 1986 are included. The high, low, opening, and closing prices, as well as the trading volume, are included for January 1987 through February 1990. In this example, we create high-low plots for a portion of the `djia` data set.

Setting up the data

Suppose we want to analyze financial data for a period of time surrounding the stock market crash of 1987. The command below uses the `positions` function to extract a subset of the `djia` time series that corresponds to the period between September 1, 1987 and November 1, 1987.

```
> dow <- djia[positions(djia) >= timeDate("09/01/87") &
+ positions(djia) <= timeDate("11/01/87"), ]
> dow

  Positions    open    high     low   close volume
 09/01/1987 2666.77 2695.47 2594.07 2610.97 193450
 09/02/1987 2606.98 2631.06 2567.76 2602.04 199940
 09/03/1987 2621.81 2642.22 2560.11 2599.49 165200
 09/04/1987 2604.11 2617.19 2556.28 2561.38 129070
 09/07/1987 2561.38 2561.38 2561.38 2561.38     NA
 09/08/1987 2551.18 2571.43 2493.78 2545.12 242880
 09/09/1987 2544.48 2570.63 2522.80 2549.27 164910
 09/10/1987 2578.13 2595.50 2549.43 2576.05 179790
 09/11/1987 2586.26 2625.96 2575.41 2608.74 178020
 09/14/1987 2624.36 2634.57 2587.85 2613.04 154380
 . . .
```

Exploratory data analysis

Create a high-low plot of the `dow` time series as follows:

1. Open the **Time Series High-Low Plot** dialog.
2. Type `dow` in the **Time Series Data** field.
3. Select `high` in the **High** list and `low` in the **Low** list.
4. Click **Apply** to leave the dialog open.

To place lines on the graph for the opening and closing prices in the `dow` time series, click on the **Data** tab in the open **Time Series High-Low Plot** dialog. Select `open` in the **Open** list and `close` in the **Close** list, and then click **Apply**. The plot is shown in Figure 6.51.

To include a panel with a barplot of the trading volume, check the **Include Barplot of Volume** box and select `volume` as the **Volume Variable**. If you prefer candlestick-style indicators instead of lines in high-low-open-close plots, click on the **Plot** tab and select **Candlestick** from the **Type** list.

It is also possible to superpose a moving average line on a high-low plot or candlestick plot. To do this, click on the **Plot** tab in the open **Time Series High-Low Plot** dialog, highlight **Specified Number** in the **Days in Average** box, and type 5 for the **Specified Number**. In our example, this computes a 5-business-day moving average of the closing stock prices in the `dow` time series. By default, the moving averages are calculated for the closing prices only; if closing values are not included in the data, moving averages are not plotted.

When you are finished experimenting, click **OK** to close the dialog.

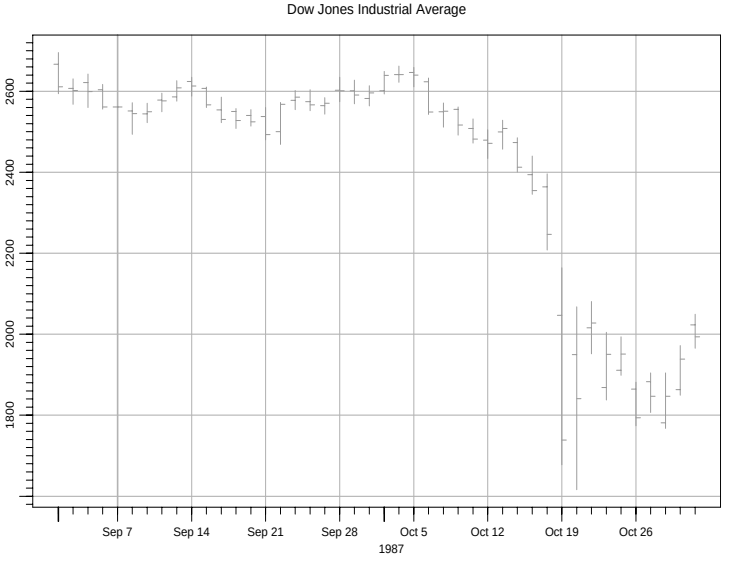


Figure 6.51: *High-low-open-close plot for a portion of the `djia` time series corresponding to the 1987 stock market crash.*

Stacked Bar Plots

A *stacked bar plot* is a chart in which multiple *y* values can represent segment heights for the bar at a single *x* value.

Creating a stacked bar plot

From the main menu, choose **Graph ▶ Time Series ▶ Stacked Bar Plot**. The **Time Series Stacked Bar Plot** dialog opens, as shown in Figure 6.52.

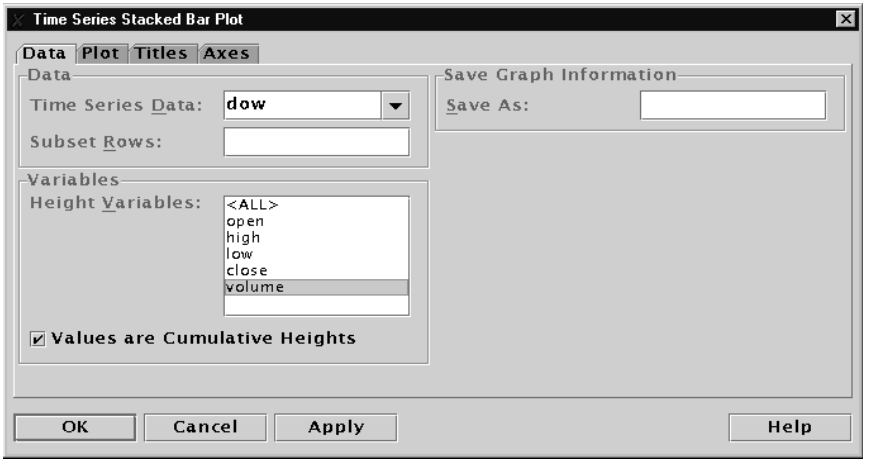


Figure 6.52: *The **Time Series Stacked Bar Plot** dialog.*

Example

In this example, we create a bar plot of the trading volume data from the `dow` time series. If you have not done so already, create the `dow` time series with the instructions given on page 205. The following steps generate the bar plot displayed in Figure 6.53.

1. Open the **Time Series Stacked Bar Plot** dialog.
2. Type `dow` in the **Time Series Data** field.
3. Select `volume` in the **Height Variables** list.
4. Click **OK**.

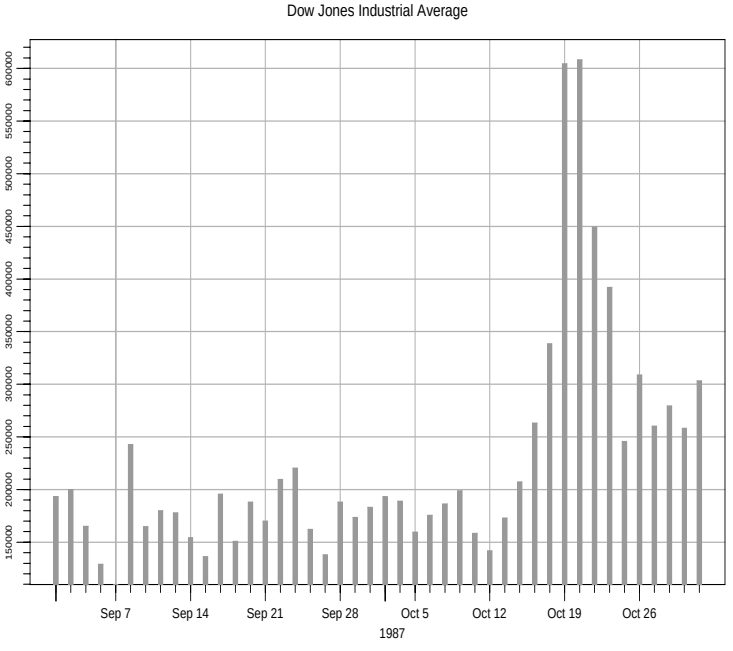


Figure 6.53: *Bar plot of the trading volume data in the* `dow` *time series.*

REFERENCES

Chambers, J.M., Cleveland, W.S., Kleiner, B. & Tukey, P.A. (1983). *Graphical Methods for Data Analysis*. Belmont, California: Wadsworth.

Cleveland, W.S. (1979). Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74: 829-836.

Cleveland, W.S. (1985). *The Elements of Graphing Data*. Monterrey, California: Wadsworth.

Cleveland, W.S. (1993). *Visualizing Data*. Murray Hill, New Jersey: AT&T Bell Laboratories.

Fisher, R.A. (1971). *The Design of Experiments* (9th ed.). New York: Hafner.

Friedman, J.H. (1984). *A Variable Span Smoother*. Technical Report No. 5, Laboratory for Computational Statistics. Department of Statistics, Stanford University, California.

Venables, W.N. & Ripley B.D. (1999). *Modern Applied Statistics with S-PLUS* (3rd ed.). New York: Springer.

WORKING WITH GRAPHICS DEVICES

7

Printing Your Graphics **212**
- Printing with PostScript Printers 212
- Using the `postscript` Function 215
- Printing with HP-GL Pen Plotters 225
- Creating PDF Graphics Files 227
- Creating Windows Metafile Graphics 227
- Creating Bitmap Graphics 227
- Managing Files from Hard Copy Graphics Devices 228
- Using Graphics from a Function or Script 229

Graphics Window Details **231**
- Basic Terminology 231
- Opening and Removing Graphics Devices 231
- The `java.graph` Graphics Window in S-PLUS 234
- The **Options** Menu and the `java.graph` Device 236
- The Motif Graphics Window in S-PLUS 244
- The **Options** Menu and the `motif` Device 247
- Available Colors Under X11 257

PRINTING YOUR GRAPHICS

One important and widespread use of S-PLUS is to produce camera-ready graphics plots for technical reports and papers. S-PLUS supports two kinds of hard copy graphics devices: PostScript laser printers and Hewlett-Packard HP-GL plotters. S-PLUS also supports publication on the World Wide Web by means of a graphics device for creating files in Portable Document Format (PDF), and popular word processing software by means of a graphics device for creating files in Windows Metafile Format (WMF) and the ability of the `java.graph` graphics device to create popular bitmap formats. These devices are discussed in the following sections. General rules for making plot files are discussed in the section Managing Files from Hard Copy Graphics Devices (page 228).

Printing with PostScript Printers

One important and widespread use of S-PLUS is to produce camera-ready graphics plots for technical reports and papers. For many S-PLUS users, that means producing graphics suitable for printing on PostScript-compatible printers.

In S-PLUS, you can create PostScript graphics using any of the following methods:

- Choose **Print** from the **Graph** menu on the `motif` windowing graphics device.
- Use the `printgraph` function with any graphics device that supports it. (The `motif` device supports `printgraph`, as do many others. See the `Devices` help file for a complete list.)
- Use the `postscript` function directly.

We discuss each of these methods in the following subsections.

If you are using `postscript` directly, the *aspect ratio* of the finished graphic is determined by the width and height, if any, that you specify, the orientation, and the paper size. If you use the other methods, by default the aspect ratio is the original aspect ratio of the device on which the graphic is originally created. For the windowing graphic device `motif`, this ratio is 8:6.32 by default. Resizing the graphics window has no effect on PostScript output created from the resized window; it retains the aspect ratio of the original, non-resized window.

Using the Print Option from the Motif Graphics Window Menu

The `motif` windowing graphics device is a convenient tool for exploratory data analysis and interactive graphics. You can easily create PostScript versions of graphics created on this device by using the **Print** option from the **Graph** menu. The behavior of this option is determined by options specified in the **Printing Options** dialog box selected from the **Options** menu. The following choices are available:

- *Method* — Should show **PostScript** selected. If not, move the pointer to the **PostScript** method and click.
- *Orientation* — Determines the orientation of the graphic on the paper. **Landscape** orientation puts the *x*-axis along the long side of the paper; **Portrait** orientation puts the *x*-axis along the short side of the paper. To choose the orientation, move the pointer to the desired choice and click.
- *Command* — A UNIX command executed when you select the **Print** option from the **Graph** menu. The default value, when **Method** is set to **PostScript**, is the command stored in the value of `ps.options()$command`. To change this command, move the pointer to this line and click to ensure the line has input focus, then edit the command.

As the default command is normally to send a file to a printer, the most common use of the **Print** option is to create immediately a hard copy of the displayed graphic. You can, however, specify a command such as the following to store the PostScript output in a named file:

```
cat > myfile <
```

Here *myfile* is any desired file name. However, the `printgraph` function, described in the next section, provides a more convenient method for creating files of PostScript output.

To choose the **Print** option from the graphics device:

1. Move the pointer to the button labeled **Graph**.
2. Click and a menu appears.
3. Drag the pointer to the **Print** option, then release the mouse button. A message appears in the footer of the graphics window telling you that the specified command has been executed.

Using the Print Option from the Java Graphics Window

The `java.graph` windowing graphics device is another convenient tool for exploratory data analysis and interactive graphics. You can easily print graphics created on this devices by using the **Print** option from the main **File** menu. The **Print** dialog has the following options:

- *Copies* Allows you to specify how many copies of the graphic to print.
- *Print to* Allows you to specify either the name of a printer, or the file name to be used to print to a file.
- *Banner Page Title* The title to appear on the banner page of your print job, if your printer is configured to print a banner page.
- *Print Command Options* Allows you to specify additional options to be sent to your print command.

As the default command is normally to send a file to a printer, the most common use of the **Print** option is to create immediately a hard copy of the displayed graphic.

Using the printgraph Function

In its simplest use, the `printgraph` function is just another way to produce immediate hard copies of graphics created on windowing or other graphics devices. Many graphics devices for use with graphics terminals and emulators, including `tek14`, support the `printgraph` function.

The default behavior of the `printgraph` function is determined by a number of environment variables. These are discussed in the section Environment Variables and printgraph (page 443). To make

`printgraph` produce PostScript output, you should make sure that the environment variable **S_PRINTGRAPH_METHOD** is set to `postscript`, or call `printgraph` directly with the argument `method="postscript"`.

S_PRINTGRAPH_METHOD determines the default value for the `method` argument to `printgraph` and specifies the type of printer for which `printgraph` produces output. Environment variables cannot be set from within S-PLUS; if you want to change an environment variable, quit S-PLUS, reset the environment variable, then restart S-PLUS.

Within your S-PLUS session, you can control the default printing behavior by using `ps.options`. We recommend that you use `ps.options` instead of environment variables whenever possible. The options that can be controlled through `ps.options` are described in the section Setting PostScript Options (page 220).

To call `printgraph` to print an immediate hard copy of the current graphic, use the following call:

```
> printgraph()
```

You can override the default method, command, and orientation with arguments to `printgraph`:

```
> printgraph(horizontal=F, method="postscript",
+ command="lpr -h")
```

Using the postscript Function

You can start the `postscript` device directly very simply as follows:

```
> postscript()
```

By default, this writes PostScript output to a temporary file using the template specified in `ps.options`. When the device is shut down, the output is printed with the command specified in `ps.options`.

You can specify many options as arguments to `postscript`; most of these are global PostScript printing options that are also used by the Print option of the windowing graphics device and by the `printgraph` function---these options are discussed in the section Setting PostScript Options (page 220). The `append`, `onefile`, and `print.it` arguments, however, are specific to calls to postscript.

The onefile argument is specified as a logical value, which defaults to TRUE. By default, when you start the postscript device explicitly, plots are accumulated into a single file as given by the file argument. If no file argument is specified, the file is named using the template specified in ps.options()$tempfile. When onefile is FALSE, a separate file is created for each plot and the PostScript file created is structured as an Encapsulated PostScript document. See the section Creating Encapsulated PostScript Files (page 218), for further details.

The append option is a logical value that specifies whether PostScript output is appended to file if it already exists. In addition to appending the new graphics, S-PLUS edits the file to comply with the PostScript Document Structuring Conventions. If append=FALSE, new graphics output writes over the existing file, destroying its previous contents.

You can use the print.it argument to specify that the graphic created on the postscript device be both sent to the printer and written to a file, as follows:

```
> postscript(file="mystuff2.ps", print.it=T)
> plot(corn.rain)
> title("A plot created with postscript()")
> dev.off()

Starting to make postscript file.
 null device
           1

> !vi mystuff2.ps

%!PS-Adobe-3.0
%%Title: (S-PLUS Graphics)
%%Creator: S-PLUS
%%For: (Rich Calaway,x240)
%%CreationDate: Thu Jul 30 21:45:21 1992
%%BoundingBox: 20 11 592 781
%%Pages: (atend)
. . .
```

Warning

If you want to both print the graphic and keep the named PostScript file, be sure that the UNIX print command does not delete the printed file. For example, on some computers, the default value of `ps.options()$command` (which is determined by the environment variable **S_POSTSCRIPT_PRINT_COMMAND**) is `lpr -r -h`, where the `-r` flag causes the printed file to be deleted. The following call to postscript replaces this default with a command that does not delete the file:

```
> postscript(file="mystuff2.ps", print.it=T, command="lpr -h")
```

Using `postscript` directly can be cumbersome, since you don't get immediate feedback on graphics produced incrementally. You can, however, build a *graphics function* incrementally, using a windowing graphics device or graphics terminal. Then, when the graphics function works well on screen, start a `postscript` device and call your graphics function. Such an approach will result in fewer hard copies for the recycling bin. For example, consider the code below, which combines into a single function the commands needed for creating a complicated graphic:

```
> usasymb.plot
function()
{
      select <- c("Atlanta", "Atlantic City", "Bismarck",
                  "Boise", "Dallas", "Denver", "Lincoln",
                  "Los Angeles", "Miami", "Milwaukee",
                  "New York", "Seattle")
      city.name <- city.name
      city.x <- city.x
      city.y <- city.y
      names(city.x) <- names(city.y) <-
                  names(city.name) <- city.name
      pop <- c(425, 60, 28, 34, 904, 494, 129, 2967, 347,
                  741, 7072, 557)
      usa()
      symbols(city.x[select], city.y[select], circles =
                  sqrt(pop), add = T)
      size <- ifelse(pop > 1000, 2, 1)
      size <- ifelse(pop < 100, 0.5, size)
      text(city.x[select], city.y[select], city.name[
                  select], cex = size)
}
```

Modifying a function containing a string of graphics commands is much easier than retyping all the commands to re-create the graphic.

Another useful technique for preparing PostScript graphics is to use PostScript screen viewers such as `ghostview`.

Creating Encapsulated PostScript Files

If you are creating graphics for inclusion in other documents, you typically want to create a single file for each graphic in a file format known as *Encapsulated PostScript*, or EPS. EPS files can be included in documents produced by many word-processing and text-formatting programs.

Documents conforming to the Adobe Document Structuring Convention Specifications, Version 3 for Encapsulated PostScript have the following first line:

```
%!PS-Adobe-3.0 EPSF-3.0
```

They must also include a BoundingBox comment. Non-EPS files have the following first line:

```
%!PS-Adobe-3.0
```

Warning

S-PLUS supports the Encapsulated PostScript file format, EPSF. It *does not* support the Encapsulated PostScript Interchange format, EPSI. EPS files created by S-PLUS do not include a preview image, so if you import an S-PLUS graphic into WYSIWYG software such as FrameMaker or Word, you will see only a gray rectangle or a box where the graphic is included.

You can use `printgraph` to produce separate files for each graphic you produce, as soon as you've finished composing it on a windowing graphics device or terminal/emulator that supports `printgraph`. You can specify the file name and orientation of the graphics file. For example, you can create the PostScript file `mystuff.ps` containing a plot of the data set `corn.rain` as follows:

```
> motif()
> plot(corn.rain)
> title("My Plot of Corn Rain Data")
> printgraph(file="mystuff.eps")
```

You can produce EPS files with direct calls to `postscript` by setting `onefile=FALSE`. To create a *single* file, with a name you specify, call `postscript` with the `file` argument and `onefile=F`:

```
> postscript(file="mystuff.eps", onefile = F, print = F)
> plot(corn.rain)
> dev.off()
```

Warning

If you supply the `file` argument and set `onefile=F` in the same call to `postscript`, you *must* turn off the device with `dev.off` after completing the first plot. Otherwise, the next plot will overwrite the previous plot, and the previous plot will be irretrievably lost.

To create a *series* of Encapsulated PostScript files in a single call to `postscript`, omit the `file` argument:

```
> postscript(onefile=F, print=F)
> plot(corn.rain)
> plot(corn.yield)

Starting to make postscript file.
Generated postscript file "ps.out.0001.ps".
```

Because `onefile` is `FALSE`, `postscript` generates a postscript file as soon as the new call to `plot` tells it that nothing more will be added to the first plot. The file "`ps.out.0001.ps`" contains the plot of `corn.rain`. A file containing the plot of `corn.yield` is generated as soon as a new call to `plot` or a call to `dev.off` closes the old plot.

```
> plot(corn.rain, corn.yield)

Starting to make postscript file.
Generated postscript file "ps.out.0002.ps".
```

You can give a series-specific naming convention for the series of files using the `tempfile` argument to `postscript`:

```
> postscript(onefile=F, print=F, tempfile="corn.####.ps")
> plot(corn.rain)
> plot(corn.yield)

Starting to make postscript file.
Generated postscript file "corn.0001.ps".
```

```
> plot(corn.rain, corn.yield)

Starting to make postscript file.
Generated postscript file "corn.0002.ps".

> dev.off()

Starting to make postscript file.
Generated postscript file "corn.0003.ps".
```

Setting PostScript Options

The behavior of the `postscript` graphics device, whether activated by the Print option from a `motif` graphics device, by a call to `printgraph`, or by a direct call to `postscript`, is controlled by options you can set with the `ps.options` function. These options allow you to control many aspects of the PostScript output, including the following:

- The name of the PostScript output file.
- The UNIX command to print your PostScript output.
- The orientation and size of the finished plot.
- Printer-specific characteristics, including paper size, number of rasters per inch, and the size of the imageable region.
- Plotting characteristics of the graphics, including the base point size for text and available fonts and colors.

Specifying the PostScript File Name

All PostScript output is initially written to a file. Unless you explicitly call the `postscript` device with the `onefile=T` argument, S-PLUS writes a separate PostScript file for each plot, in compliance with the Encapsulated PostScript Document Structuring Conventions. You can specify the file name for the output file using the `file` argument to `postscript` or `printgraph`, or provide a template for multiple file names using the PostScript option `tempfile`, which defaults to `"ps.out.####.ps"`. You can specify this option as an argument to the `printgraph`, `postscript`, and `ps.options` functions. The template you specify must include some # symbols, as in the default. S-PLUS replaces the first series of these symbols that it encounters with a sequential number of the same number of digits in the generated file names. For example, if you have a project involving the

`halibut` data, and you know your project will use fewer than 1000 graphics files, you can set the `tempfile` option as follows to use the name of your data set:

```
> ps.options(tempfile="halibut.###.ps")
```

Specifying a Printer Command

What happens to the file after it is created is determined by the `command` option. The `command` option is a character string specifying the UNIX command used to print a graphic. If `file` is specified (and is neither a template nor an empty string), the `command` option must be activated by some user action, either choosing the Print option from a windowing graphics device, specifying `print=TRUE` in the `printgraph` function, or specifying `print.it=TRUE` in the `postscript` function.

The default for `command` is the value of the environment variable **S_POSTSCRIPT_PRINT_COMMAND** .

Specifying Plot Orientation and Size

You specify the plot orientation with the `horizontal` option: `TRUE` for landscape mode (*x*-axis along long edge of paper), `FALSE` for portrait. Most figures embedded in documents should be created in portrait mode, because that is the usual orientation of documents. The default is the orientation specified by the **S_PRINT_ORIENTATION** , which by default is set to `TRUE`, that is, landscape mode. If you specify an orientation with your graphics window's Options Printing menu, that specified orientation is taken to be the default.

You specify the plotting region, in inches, with the `width` (the *x*-axis dimension) and `height` (*y*-axis dimension) options. Thus, to create graphics for inclusion in a manual, you might specify the following options:

```
> ps.options(horizontal=F, width=5, height=4)
```

The default value for `width` and `height` are determined by the printer's imageable region, as described in the next subsection.

Specifying Printer Characteristics

PostScript can describe pages of virtually any size, but it does little good to create enormous page descriptions if you don't have an output device capable of printing them. Most PostScript printers have remarkably similar characteristics, so you may not have to change the options that specify them. For example, in the United States, most printers default to `"letter"` (8 1/2 x 11) paper. Among the options that you can specify for your printer, the `paper` option is the most important. The `paper` argument is a character string; most standard ANSI and ISO paper sizes are accepted. Each paper size has a specific *imageable region*, which is the portion of the page on which the printer can actually print. This region can vary slightly depending on the printer hardware, even for paper of the same size. The imageable region determines the default values for the `width` and `height` options.

Specifying Plotting Characteristics

The PostScript options that have the greatest immediate impact on what you see are those affecting the PostScript graphic's plotting characteristics. These options include the following:

- `fonts` A vector of character strings specifying all available fonts.
- `colors` A numeric vector or matrix assigning actual colors to the color numbers used as arguments to graphics functions. This option is discussed in more detail in the next section.
- `image.colors` Same as `colors`, but for use with the `image` function.
- `background` A numeric vector giving the color of the background, as in `colors.background`, can also be a single number that is used as an index to the `colors` argument if it is positive or, if it is negative, specifies no background at all.

Creating Color PostScript Graphics

Creating PostScript graphics in color is no more difficult than creating color graphics on your windowing graphics device. With the `xgetrgb` function, you can copy the color map from the current `motif` device and use it for PostScript output. The following steps show how to print graphics from a `motif` window to a PostScript printer using the same color map.

1. Start the graphics window:

   ```
   > motif()
   ```

2. Set the color scheme using the **Color Scheme** dialog box, accessible from the **Options** menu. See the section The Options Menu and the motif Device (page 247) for complete details.

3. Plot the graphic in the graphics window:

   ```
   > image(voice.five)
   ```

4. Capture the colors from the device using `xgetrgb`:

   ```
   > my.colors <- xgetrgb(type="images")
   ```

   The `type` argument to `xgetrgb` should be appropriate for the type of graph being reproduced. Here, we use `type="images"` because we want the colors used to produce an image plot. The default type is `"polygons"`, which is appropriate for bar plots, histograms, and pie charts, and is usually also suitable for scatter plots and line plots such as time series plots. Other valid types are `"lines"`, `"text"`, and `"background"`.

5. Send the color specification to update the graphics window's printer options:

   ```
   > ps.options.send(image.colors=my.colors)
   ```

   The `image.colors` argument assigns colors for image plots. Use the `colors` argument to assign colors for all other plots. Use the `background` argument to specify the background color.

You can, of course, use the results of `xgetrgb` as arguments without first assigning them to an S-PLUS object, as is shown below:

```
> ps.options.send(image.colors=xgetrgb("images"),
+ colors=xgetrgb("lines"),
+ background = xgetrgb("background"))
```

6. Select the **Print** button to print the colored graphic.

To create color graphics with the `postscript` function, you follow essentially the same steps, as in the following example:

1. Start the graphics window:

```
> motif()
```

2. Set the desired color scheme using **Options ▶ Color Scheme** from the `motif` menu.
3. Capture the colors from the device using `xgetrgb` and specify the captured colors as the PostScript color scheme using `ps.options`:

```
> ps.options(colors = xgetrgb("colors"),
+ background = xgetrgb("background"))
```

4. Start the postscript device using the `postscript` function:

```
> postscript(file = "colcorn.ps")
```

5. Plot the graphic; the following commands produce a plot with three different colors:

```
> plot(corn.rain, corn.yield, type="n")
> points(corn.rain, corn.yield, col=2)
> title(main="A plot with several colors", col=3)
```

6. Turn off the postscript device:

```
> dev.off()
```

Printing with HP-GL Pen Plotters

The `hpgl` graphics device translates your S-PLUS plotting commands into commands that can be read by pen plotters that accept the Hewlett-Packard HP-GL instruction set. To start the `hpgl` graphics device, type:

```
> hpgl(file = "file")
```

where *file* is a file name specifying where to write the plotting commands. When the `hpgl` device is the current graphics device, no graphics appear on your screen.

The following arguments may be supplied to the `hpgl` function:

- `width` — Determines the width of the *x*-axis dimension (in inches). The default value is `10`.
- `height` — Determines the height of the *y*-axis dimension (in inches). The default value is `7.25`.
- `ask` — Determines whether you are prompted by `"GO?"` prior to advancing to a new frame. Possible values are `TRUE` and `FALSE`. The default value is the opposite of the value of `auto`.
- `auto` — Determines whether the device can automatically advance the paper. Possible values are `TRUE` and `FALSE`. The default value is `FALSE`.
- `color` — Determines the degree of color-plotting support provided by the device. See the help file for details.
- `speed` — Determines maximum allowed axis-pen velocity. See the help file for details.

- `rotated` Determines whether the *x*-axis lies along the long side of the paper (landscape mode) or the short side of the paper (portrait mode). Possible values are `TRUE` (portrait mode) and `FALSE` (landscape mode). The default value is `FALSE`.

- `file` Determines the name of the file that the HP-GL commands are stored in. By default, the commands are sent to your terminal.

- `hw.control` Determines whether hardware control escape sequences are to be included. These escape sequences may be unnecessary depending on how the output is to be used. For example, if the output will be imported into another software package, it may help to set `hw.control` to `FALSE`. The default is `TRUE`.

To use the `hpgl` graphics device, follow these steps:

1. Type the `hpgl` command along with any arguments you want to specify. For example, use the `file` argument to send your graphics output to a file.
2. Type your S-PLUS graphics commands.

For example, the following commands start the `hpgl` graphics device with the `file` argument to name the output file, then make a scatter plot and time series plot, using `dev.off` to append the second plot to the file and turn off the `hpgl` device. After sending the files to the plotter, we remove them:

```
> hpgl(file="hpgl.com")
> plot(corn.rain, corn.yield)
> ts.plot(lynx)
> dev.off() # Append the last plot to hpgl.com
> ! lpr -P hpgl hpgl.com
> ! rm hpgl.com
```

In this example, two plots are written to the file `hpgl.com`. We then escape to the UNIX shell and issue the **lpr** command to send the file to the plotter. (The command for sending your file to the plotter may be different for your system.) Finally, we escape to the UNIX shell and issue the **rm** command to remove the file.

Creating PDF Graphics Files

The Portable Document Format (PDF) is a popular electronic publishing format closely related to PostScript. You can create PDF graphics files in S-PLUS using the `pdf.graph` graphics device. You can create a PDF graphics file simply by calling `pdf.graph` with the desired output file name:

```
> pdf.graph("mygraph.pdf")
> plot(corn.rain, corn.yield, main="Another corny plot")
> dev.off()
```

Once you've created your PDF graphics, you can view them using Adobe's Acrobat Reader (available on most personal computers and some UNIX platforms). See the `pdf.graph` help file for more details.

Creating Windows Metafile Graphics

The Windows Metafile is a popular graphics format for inclusion for Windows-based word processors and spreadsheets such as Microsoft Word and Excel. You can create WMF graphics files in S-PLUS using the `wmf.graph` graphics device. You can create a WMF graphics file simply by calling `wmf.graph` with the desired output file name:

```
> wmf.graph("mygraph.wmf")
> plot(corn.rain, corn.yield, main="Another corny plot")
> dev.off()
```

For more details, see the `wmf.graph` help file from within S-PLUS 6.0.

Creating Bitmap Graphics

Bitmap graphics are popular because they are easy to include into most word processing software. They are not recommended for most statistical graphics, because they tend to have lower resolution than normal S-PLUS vector graphics, such as those produced on screen by the `java.graph` or `motif` devices, or in files by the `postscript`, `pdf.graph`, or `wmf.graph` devices. Bitmaps can be useful for image graphics, such as those produced by the `image` function.

To create a bitmap graphic, start `java.graph` with a `file` argument and, if necessary, a `format` argument. The supported format arguments are `"JPEG"`, `"BMP"`, `"PNG"`, `"PNM"`, and `"TIFF"`. `"JPEG"` is the default. For example, to create a JPEG image of the `voice.five` data, use `java.graph` as follows:

```
> java.graph("voice.jpeg", format="JPEG")
> image(voice.five)
> dev.off()
```

Managing Files from Hard Copy Graphics Devices

With all hard copy graphics devices, a plot is sent to a plot file not when initially requested, but only after a subsequent high-level graphics command is issued, a new frame is started, the graphics device is turned off, or you quit S-PLUS. To write the current plot to a plot file (assuming you have started the graphics device with the appropriate file option), you must do one of the following:

- Make another plot (assuming a single figure layout).
- Call the function `frame` (again, assuming a single figure layout).
- Call the function `dev.off` to turn off the current graphics device.
- Call the function `graphics.off` to turn off all of the active graphics devices.
- Quit S-PLUS.

Once you have created a graphics file, you can send it to the printer or plotter without exiting S-PLUS by using the following procedure:

1. Type `!` to escape to UNIX.
2. Type the appropriate printing command, and then the name of the file.
3. Type a carriage return.

To remove graphics files after sending them to the plotter without exiting S-PLUS:

1. Type `!` to escape to UNIX.
2. Type **rm *file***, where ***file*** is the name of the graphics file you want removed.
3. Type a carriage return.

Using Graphics from a Function or Script

Most experienced users of S-PLUS use a function or *script* to construct complicated plots for presentation or publication. This method lets you use the `motif` display device to preview the plots on your screen, and then, once you are satisfied with your plots, send them to a hard copy device without having to re-type the same plotting commands.

Note

Direct use of a hard copy device ensures the best hard copy output.

To use this method using an S-PLUS function, follow these steps:

1. Put all the S-PLUS commands necessary to create the graphs into a function in S-PLUS (say `plotfcn`) using `fix`. Do *not* include commands that start a graphics device.
2. In S-PLUS, start a graphics device, then call your function:

```
> motif()
> plotfcn()
```

Note

If you are creating several plots on separate pages, you may want to set the graphics parameter `ask` to `TRUE` before calling your plotting function. In this case, the sequence of steps is:

```
> motif()
> par(ask = T)
> plotfcn()
```

3. View your graphs. If you want to change something, use `fix` to modify your plotting function.
4. Once you are satisfied with your plots, start a hard copy graphics device, call your function, and then turn the hard copy graphics device off:

```
> postscript()
> plotfcn()
> dev.off()
```

5. Save your function containing graphics commands if you will need to reproduce the plots in the future.

To use this method using a script, follow these steps:

1. Put all the S-PLUS commands necessary to create the graphs into a file outside of S-PLUS (say `plotcmds.asc`) using an editor (e.g., `vi`). Do *not* include commands that start a graphics device.
2. In S-PLUS, start a graphics device, then use `source` to execute the S-PLUS commands in your file:

```
> motif()
> source("plotcmds.asc")
```

3. View your graphs. If you want to change something, edit your file with an editor.
4. Once you are satisfied with your plots, start a hard copy graphics device, source your plotting commands, and then turn the hard copy graphics device off:

```
> postscript()
> source("plotcmds.asc")
> dev.off()
```

5. Save your file of graphics commands if you will need to reproduce the plots in the future.

GRAPHICS WINDOW DETAILS

This section describes, in detail, how to use the `java.graph` and `motif` graphics devices. The `java.graph` device is available only with Java-enabled versions of S-PLUS. The `motif` device is available only on machines that run either the X Window System, Version 11 (X11). Both devices are available on all UNIX platforms.

Both devices let you interactively change the color specifications of your plots and immediately see the result and also interactively change the specifications that are used to send the plot to a printer.

In this section, we assume you are familiar with your particular window system. In particular, we assume you know how to start your window system and set your display so that X11 applications can display windows on your screen. For further information on a particular window system, consult your system administrator or the following references:

- Quercia, V. and O'Reilly, T. (1989). *X Window System User's Guide*. Sebastopol, California: O'Reilly and Associates.
- Quercia, V. and O'Reilly, T. (1990). *X Window System User's Guide, Motif Edition*. Sebastopol, California: O'Reilly and Associates.

Basic Terminology

In this section, we refer to the window in which you start S-PLUS as the *S-PLUS window*. The window that is created when you start a windowing graphics device from the S-PLUS window is called the *graphics window*.

Opening and Removing Graphics Devices

To open a `java.graph` graphics device, type:

```
> java.graph()
```

at the S-PLUS prompt. (The `java.graph` device is also started automatically in the Java GUI version of S-PLUS if no other graphics device is open when you ask S-PLUS to evaluate a high-level plotting function.)

To open a motif graphics device, type:

```
> motif()
```

at the S-PLUS prompt. (The motif device is also started automatically in both the Java-enabled and Java-disabled command-line versions of S-PLUS if no other graphics device is open when you ask S-PLUS to evaluate a high-level plotting function.)

To remove a graphics window without quitting S-PLUS, use the function dev.off or graphics.off.

Warning

Do not destroy the motif *graphics window by using a window manager menu!* If you remove a motif window in this way, S-PLUS will not know that the graphics device has been removed. Thus, this graphics device will still appear on the vector returned by dev.list, but if you try to send plot commands to it you will get an error message. If you do accidentally remove the motif window with a window manager menu, use the dev.off function to tell S-PLUS that this device is no longer active. The java.graph device correctly shuts down if you close it using the standard window system tools.

Example

As you try out the various features of the motif and java.graph devices, you can use the following S-PLUS commands to generate an easily-reproducible graphic:

```
> plot(corn.rain, corn.yield, type="n",
+       main="Plot Example")

> points(corn.rain, corn.yield, pch="*", col=2)

> lines(lowess(corn.rain, corn.yield), lty=2, col=3)

> legend(12, 23, c("Color 1", "Color 2", "Color 3"),
+      pch=" * ", lty=c(1, 0, 2), col=c(1, 2, 3))
```

Note that in the call to legend there is a space before and after the * in the argument pch="*". The plot generated by these commands is shown in figure 7.1.

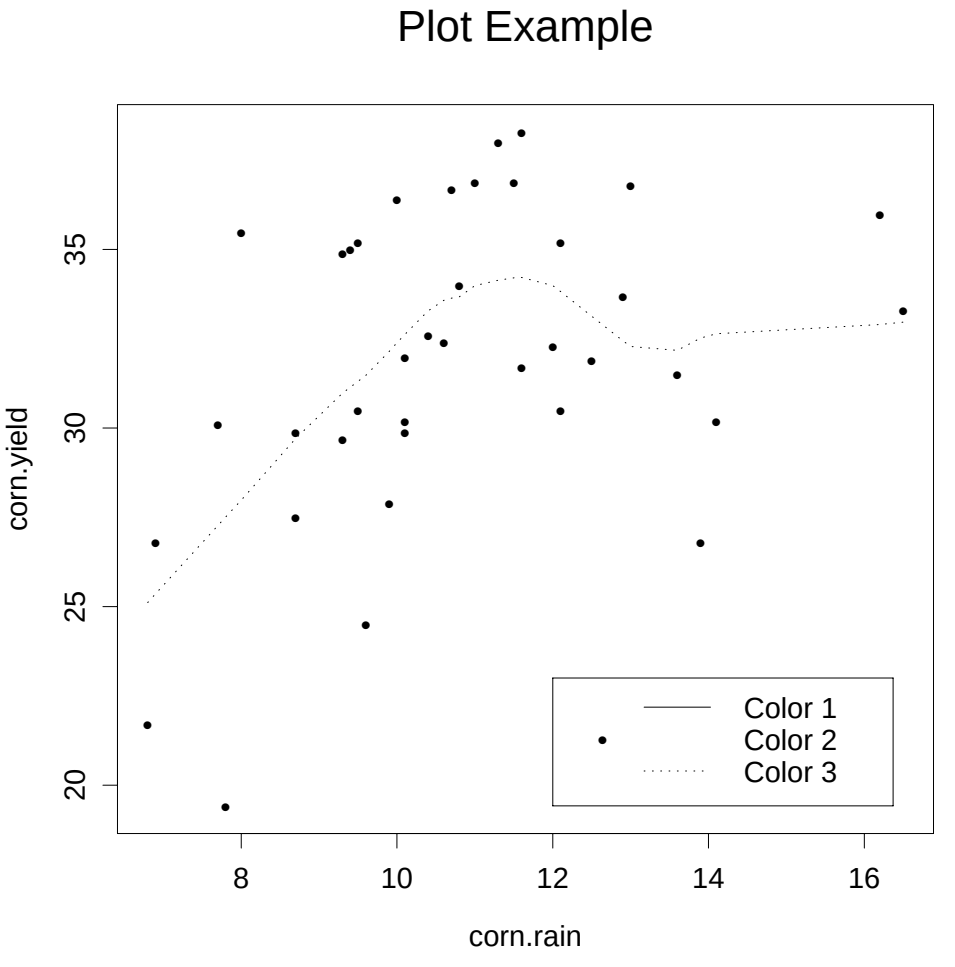


Figure 7.1: *Plot example.*

By default, the color of the title, legend box, axis lines, axis labels, and axis titles are color 1. We have specified the points to have color 2, and the dashed line representing the smooth from the `lowess` command to have color 3. Although we can't show you the difference in the colors in Figure 7.1, you will see the differences in your graphics window.

The java.graph Graphics Window in S-PLUS

Figure 7.2 shows what the `java.graph` graphics window looks like when you first start the `java.graph` device from the S-PLUS Java GUI. If you run `java.graph` in the Java-enabled command-line version of S-PLUS instead, the window also includes a menu bar with **File**, **View**, and **Options** menus, which contain a subset of the options available in the Java GUI. The elements of the graphics window are listed below.

- *Title bar* — Contains a title of the form **Graph Window *n***, the **Minimize** button, the **Maximize** button, and the **Close** window button.
- *Page* — Area where S-PLUS displays any graphs that you create while the `java.graph` graphics device is active. A `java.graph` device can have multiple pages.
- *Tab bar* — Area showing the page tabs; use this to quickly move between pages.
- *Resize Borders* — Used to change the size of the window.

If you right-click in the **Tab** bar, you obtain a menu with the following options:

- *Zoom In* — Expand the graph.
- *Zoom Out* — Shrink the graph.
- *Zoom to Rectangle* — Expand the graph so that the contents of a specified rectangle fills the window. Specify the rectangle by left-clicking in a corner, dragging the mouse, and then releasing it in the opposite diagonal corner. You must define the rectangle *before* choosing **Zoom to Rectangle** for the graph to be properly resized.
- *Fit in Window* — Resize the graph so that it fits completely within its window.

- *Set Graph Colors...* Open the **Set Graph Colors** dialog. This dialog is discussed in detail later in this section.

- *Graph Options...* Open the **Graph Options** dialog. This dialog is discussed in detail later in this section.

- *Page Properties...* Open the **Page Properties** dialog, which allows you to specify a page title and page tag. To use this dialog, right-click on the *tab* of the page that you want to modify; if you select **Page Properties** after simply right-clicking on the **Tab** bar, no dialog appears.

- *Insert Page* Insert a new page after the selected tab. To use this option, right-click on the tab of the page that should precede the new page, and choose **Insert Page**. If this tab is the currently active one, the new page is made active.

- *Delete Page* Delete the selected tab and its associated page. To use this option, right-click on the tab of the page that should be deleted, and choose **Delete Page**.

- *Clear Page* Clear the selected page. To use this option, right-click on the tab of the page that should be cleared, and choose **Clear Page**.

- *Delete All Pages* Delete all pages in the current graphics window.

Note that if you resize a `java.graph` window, the graph region resizes but maintains the same height-to-width ratio, adding gray borders on the sides if necessary. Printing a graph from a `java.graph` window also maintains the aspect ratio, expanding as much as possible to fill the page.

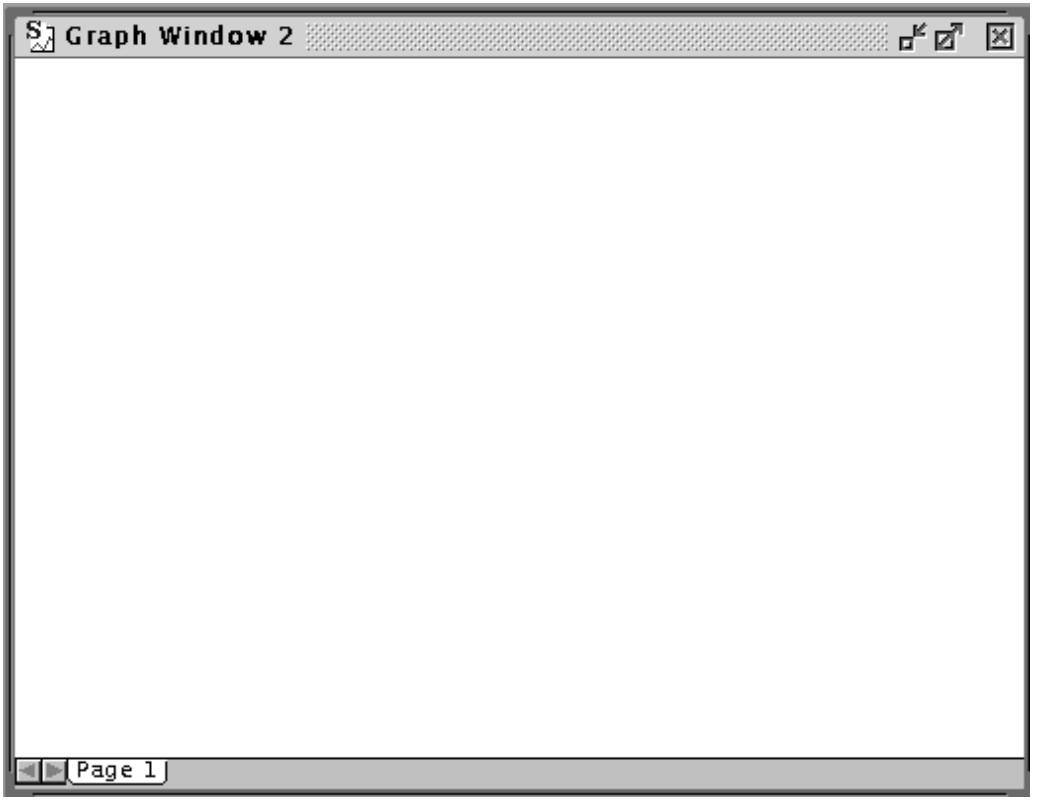


Figure 7.2: *The `java.graph` window.*

The Options Menu and the java.graph Device

If you are running `java.graph` in the S-PLUS Java GUI, the main **Options** menu contains options specific to the `java.graph` device. If you run `java.graph` in the Java-enabled command line version of S-PLUS, the **Options** menu in the graphics window is used to set options used by all `java.graph` devices. Move the pointer to the **Options** menu and click to see two menu items displayed: **Set Graph Colors...** and **Graph Options....** The ellipses (three trailing periods) indicate that dialog boxes will appear if you choose these items.

The Set Graph Colors Dialog Box

Use the **Set Graph Colors** dialog box to set the color scheme for the selected `java.graph` window. The **Set Graph Colors** dialog box is a powerful feature of the `java.graph` windowing graphics device: it lets you change the colors in your plot interactively and immediately see the results. Figure 7.3 shows an example of the **Set Graph Colors** dialog box.

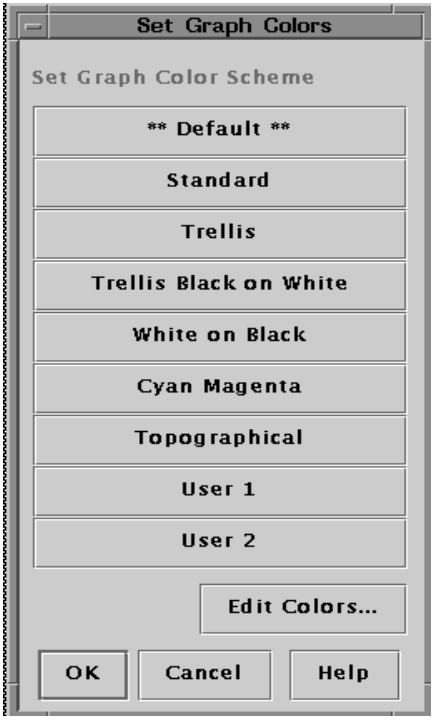


Figure 7.3: *The `java.graph` **Set Graph Colors** dialog box.*

When you first call up the **Set Graph Colors** dialog box, the pane contains:

- selection buttons for each of the available color schemes
- a button marked **Edit Colors**.
- a button marked **OK**.
- a button marked **Cancel**.
- a button marked **Help** (available in the Java GUI only).

The Help Button

The **Help** button is located in the lower right-hand corner of the **Set Graph Colors** dialog box. Click on this button to view the help window for this dialog box, which contains essentially the information presented here. Click on the **Close** button in the **Help** pop-up window to make it disappear once you are done with it. The **Help** button is available in the Java GUI only.

Available Color Schemes

The following color schemes can be selected from the **Set Graph Colors** dialog:

- *Default* — The default color scheme, used when graphs are first created in a `java.graph` window. Initially, the **Default** color scheme is the **Standard** color scheme, which uses a white background with a palette of darker colors for lines. However, you can customize this so that any color scheme appears by default in your graphs.
- *Standard* — The standard color scheme, which uses a white background with a palette of darker colors for lines. Initially, this is used as the **Default** color scheme; it is available mainly so you can recover the initial **Default** color scheme after temporarily customizing it for your graphics.
- *Trellis* — The Trellis color scheme. This uses a gray background, mostly pastel line colors, and the cyan-magenta color scale for images.
- *Trellis Black on White* — A grayscale color scheme for Trellis, which uses a white background, various shades of gray for lines, and a grayscale for images.
- *White on Black* — A grayscale color scheme with a black background, white and various shades of gray for lines, and a grayscale for images.

- *Cyan Magenta* — A color scheme with a white background, an assortment of line colors, and a cyan-magenta color scale for images. Unlike the other cyan-magenta color scales described, this one scales through black rather than through white.

- *Topographical* — A color scheme similar to Cyan Magenta, except with image colors chosen to provide a reasonable representation of topographical data.

- *User 1, User 2* — Color schemes similar to the standard color scheme, these are intended for further customization by end users.

Selecting a Different Color Scheme

To select a different color scheme, move the pointer to one of the color scheme names in the **Set Graph Colors** dialog and click. The name of the newly chosen color scheme is highlighted, and the selected `java.graph` window shows the chosen color scheme. This, however, is temporary. To make the change permanent, you must click on the **OK** button. If you click **Cancel**, the previous color scheme is restored.

Editing Colors

Each color scheme consists of four editable parts: a *name*, a *background color*, a set of *line colors*, and a set of *image colors*. To view the colors in a color scheme, click on **Edit Colors** in the **Set Graph Colors** dialog. This brings up the **Edit Graph Colors** dialog shown in Figure 7.4, which displays the currently selected color scheme.

Use the top of the **Edit Graph Colors** dialog to edit individual colors within a color scheme. To edit the background color, click the **Edit Background Color** button in the **Edit Graph Colors** dialog. To edit colors in the **Line Colors** or **Image Colors** palettes, click on a color rectangle, then select either the **Edit Selected Line Color** button or **Edit Selected Image Color** button as appropriate. The currently selected color is surrounded by a red border. You can select multiple consecutive colors by dragging the mouse over the desired colors; the red border appears around all selected colors.

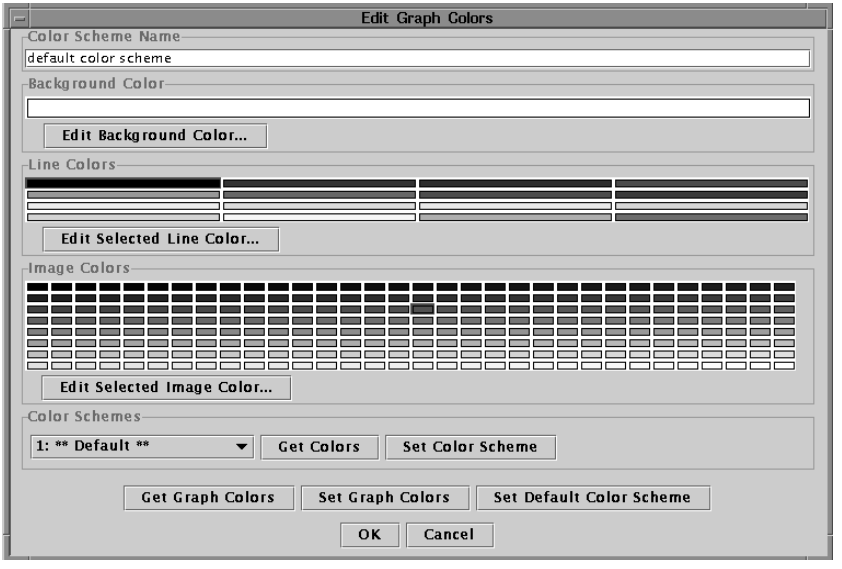


Figure 7.4: *The **Edit Graph Colors** dialog.*

The three buttons labeled **Edit *xxx* Color** in the **Edit Graph Colors** dialog bring up identical dialogs, titled **Edit *xxx* Color**. The **Edit Image Color** dialog is shown in Figure 7.5.

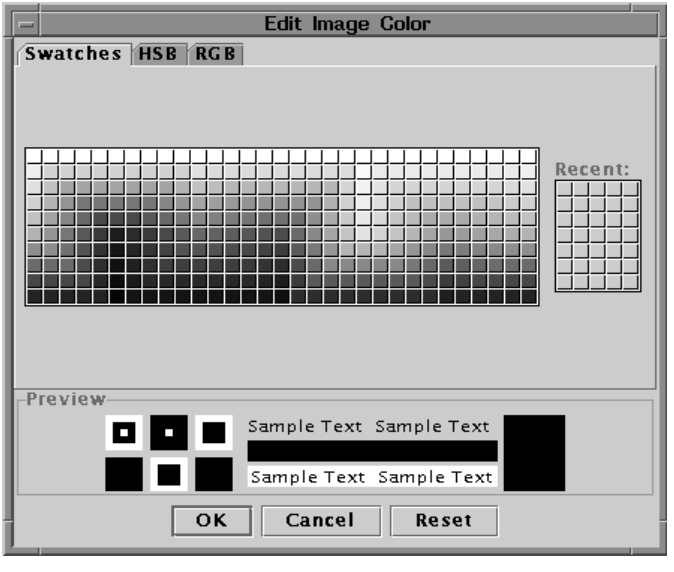


Figure 7.5: *The **Edit Image Color** dialog.*

Each dialog has three tabs: **Swatches**, **HSB**, and **RGB**. The three tabs provide alternative but equivalent methods for modifying your colors. The **Swatches** tab is the easiest to use: simply select a color from the palette of colors, examine the **Preview** section to see if it has the effect you're looking for, then click **OK**.

The **HSB** tab lets you specify colors using the HSB model (Hue-Saturation-Brightness) used by the PostScript page description language. Use this tab if you have an HSB color map you are trying to match exactly in your `java.graph` device. You can either specify the HSB values exactly, using the H, S, and B text fields, or relatively, by using the pointer on the color bar. The H values are drawn from a color wheel, so H accepts the values 0 to 359. The S and B values are percentages with 0 being none of the quality and 100 being full value. The color bar can select values for any of the three qualities, depending on which of the H, S, and B radio buttons is active. The H color bar appears as a rainbow of colors. The S color bar is the selected color shown with varying saturation, from white (no saturation) to full intensity color. The B color bar shows the amount of light in the color, from none (black) to full. The **HSB** tab also shows you, for your information only, the associated RGB color of the current HSB settings.

The **RGB** tab allows you to specify colors using the standard Red-Green-Blue color model. Use the sliders or the text fields to describe the appropriate RGB values.

Use the bottom of the **Edit Graph Colors** dialog to manipulate color schemes and graph colors, as follows:

- *Color Schemes popup list*
  Use this list to select one of the known color schemes. Note that selecting a color scheme does not update the colors in the **Edit Graph Colors** dialog.

- *Get Colors*
  Retrieves the colors from the color scheme selected in the popup list and update the displayed colors.

- *Set Color Scheme* Sets the color scheme selected in the popup list to the displayed colors. This setting is temporary until you click **OK**. If you click **Cancel**, the previous colors are restored.

- *Get Graph Colors* Retrieves the colors from the color scheme of the selected graph. This essentially restores the initial colors in the dialog, since the colors from the selected graph's color scheme are shown when the dialog first opens.

- *Set Graph Colors* Sets the color scheme of the selected graph window to be the current palette of colors. You can use this option to temporarily test combinations of colors on an active graph. To commit color changes made with this option, click the **OK** button; if you click **Cancel** , all changes are lost.

- *Set Default Color Scheme* Sets the default color scheme to the displayed colors. This is equivalent to selecting the default color scheme in the **Color Schemes** popup list, then clicking **Set Color Scheme**.

Use the **OK** button to apply the current changes and exit the dialog. Use the **Cancel** button to restore the previous state and close the dialog.

The Graph Options Dialog

The second graph menu item under the **Options** menu is labeled **Graph Options**. This brings up the **Graph Options** dialog, shown in Figure 7.6. Use the radio buttons under **New Plot Action** as described below to specify how the graphics window should respond to *clear commands*. Clear commands are generated whenever S-PLUS attempts to create a new high-level graphic.

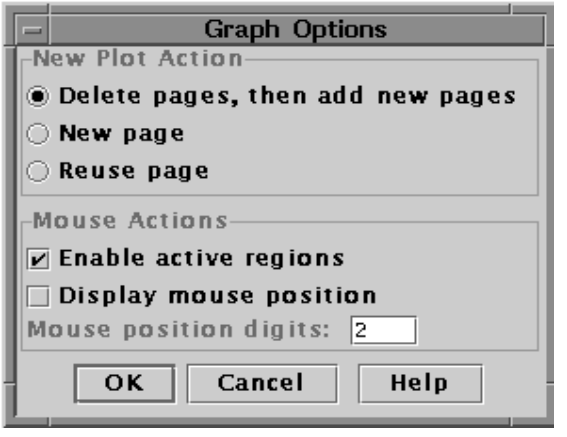


Figure 7.6: *The **Graph Options** dialog.*

- *Delete pages, then add new pages*
  The first time that a clear command is issued to a `java.graph` device within a top-level expression, all existing pages in the window are deleted and a new "Page 1" is created. Additional clear commands within the top-level expression create additional pages. In this mode, graphics exist in the device only until a new top-level graphics expression replaces them.

- *New page*
  Whenever a clear command is issued, create a new page. Use this mode to keep all your graphics for a session within a single `java.graph` device.

- *Reuse page*
  Whenever a clear command is issued, clear the current page. In this mode, functions that display multiple plots will end up displaying just the last one.

Use the check boxes under **Mouse Actions** as follows:

- *Enable active regions* Select this checkbox to enable active regions created with `java.identify` to be highlighted as the mouse passes over them and their associated actions to be performed when the mouse is clicked in the region. The default is selected.

- *Display mouse position* Select this checkbox to display *x-y* coordinates of the mouse in the upper-right corner of the graph window. The text field immediately following, labeled **Mouse position digits**, allows you to specify the number of decimal digits to use when displaying mouse coordinates.

The Motif Graphics Window in S-PLUS

Figure 7.7 shows what the Motif graphics window looks like when you first start the S-PLUS `motif` windowing graphics device. The features of this window are listed below.

- *Title bar* Contains the window **Menu** button, the title **S-PLUS**, the **Minimize** button, and the **Maximize** button.

- *Menu Bar* Contains three menu titles: **Graph**, **Options**, and **Help**. The **Help** menu title produces a pop-up window, rather than a menu, when you select it.

- *Pane* Area where S-PLUS displays any graphs that you create while the `motif` graphics device is active.

- *Footer* Area where S-PLUS puts status or error messages concerning the graph you have created.

- *Resize Borders* Used to change the size of the window.

The Help Menu

The **Help** menu title appears at the far right side of the menu bar. Move the pointer to this menu title and click to call up a help pop-up window. This help window contains a condensed version of the `motif` help file. Click on the **Close** button in this pop-up window to make this window disappear once you have finished with it.

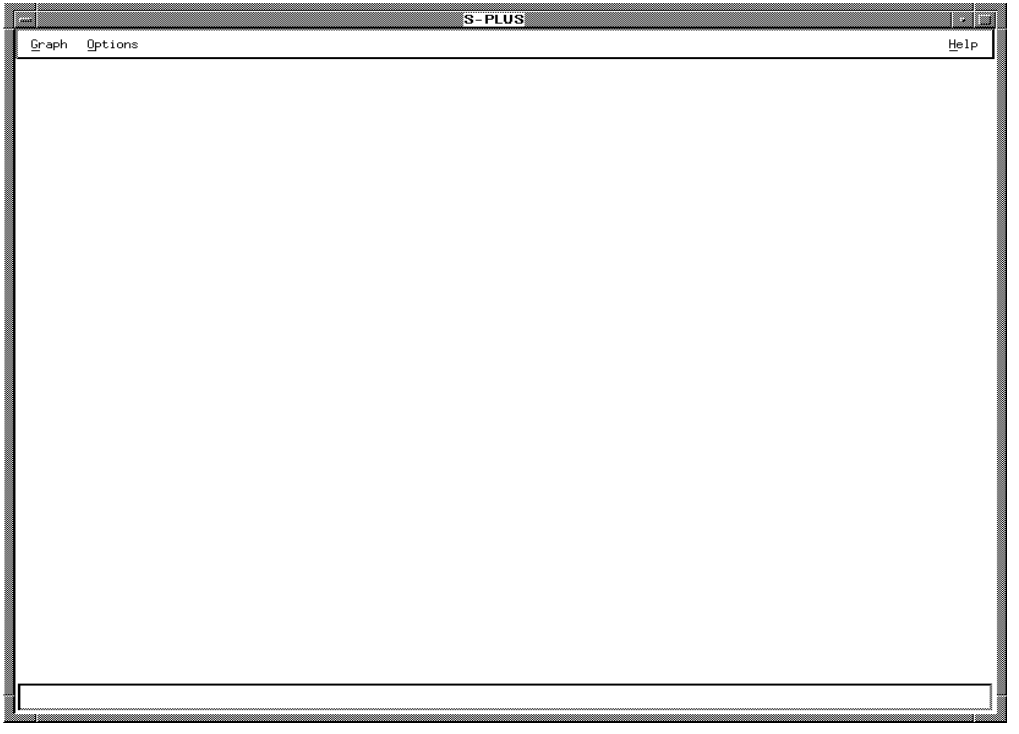


Figure 7.7: *The `motif` window.*

The Graph Menu

The first menu title in the menu bar of the graphics window is the **Graph** menu title. Move the pointer to this title and click to call up a menu with the following items:

- *Redraw* — Redraws the graph that appears in the pane of the graphics window.
- *Copy* — Creates a copy of the current graphics window, as shown in Figure 7.8. The copy has a title bar, a menu bar, a pane, and a footer, just like the original. The title in the

title area is **S-PLUS: Copy**. The menu bar in a copy of the graphics window does *not* contain an **Options** menu title, only the **Graph** and **Help** menu titles.

- *Print* — Converts the current plot in the graphics window to either a PostScript or LaserJet file and then sends this file to your printer. Choosing Print is *not* equivalent to typing the `printgraph()` command in the S-PLUS window. The `printgraph` command uses S-PLUS environment variables to determine printing defaults, whereas **Print** uses the specifications shown in the **Printing...** dialog box.

When you select **Print**, a message is displayed in the footer of the graphics window telling you what kind of file was created and the command that was used to route this file to the printer. See the section The Options Menu and the motif Device (page 247) for a description of how to set the defaults for printing.

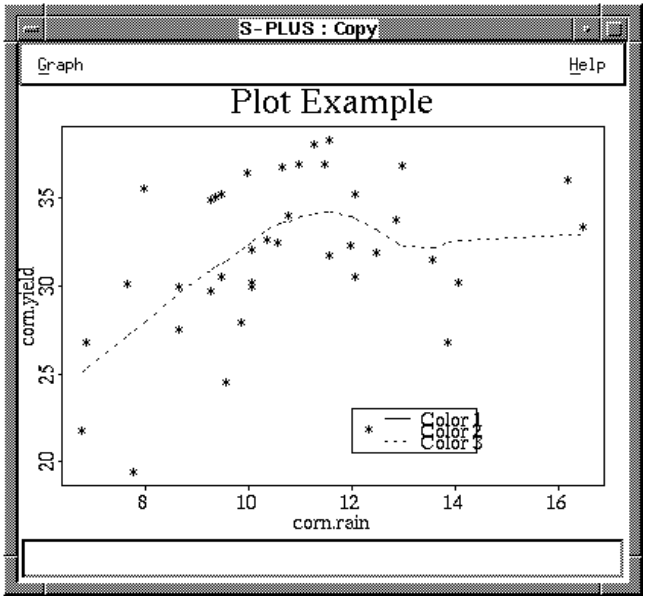


Figure 7.8: *A copy of the* `motif` *graphics window.*

The Options Menu and the motif Device

The **Options** menu title is the second menu title in the menu bar of the `motif` graphics window. Move the pointer to this title and click to see two menu items displayed: **Color Scheme...** and **Printing....** The ellipses (three trailing periods) indicate that dialog boxes will appear if you choose these items.

The Color Scheme Dialog Box

The **Color Scheme** dialog box is a powerful feature of the `motif` windowing graphics device: it lets you change the colors in your plot interactively and immediately see the results. Figure 7.9 shows an example of the **Color Scheme** dialog box. This window has a title bar with a window menu button and the title **S-PLUS Color Scheme Editor**.

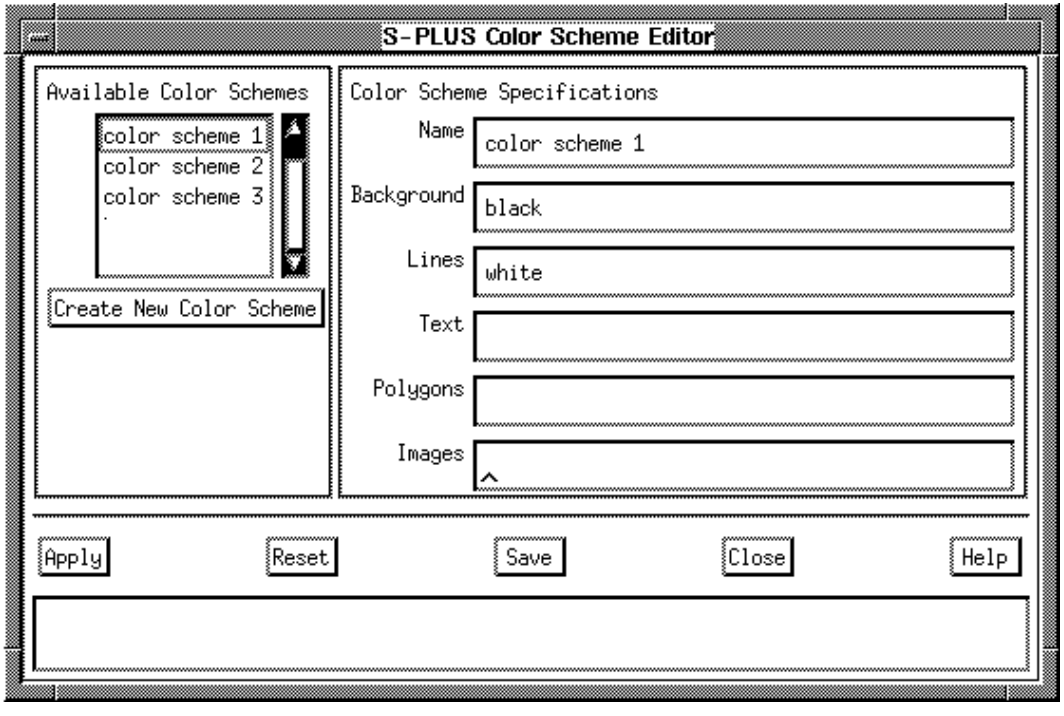


Figure 7.9: *The motif **Color Scheme** dialog box.*

When you first call up the **Color Scheme** dialog box, the pane contains:

- The **Available Color Schemes** menu.
- The **Color Scheme Specifications** editor showing the specifications for the default color scheme.
- A button marked **Create New Color Scheme**.
- A button marked **Apply**.
- A button marked **Reset**.
- A button marked **Save**.
- A button marked **Close**.
- A button marked **Help**.

The Help Button

The **Help** button is located in the lower right-hand corner of the **Color Scheme** dialog box. Click on this button to view a pop-up help window for this dialog box. Click on the **Close** button in the **Help** pop-up window to make it disappear once you are done with it.

The Color Scheme Specifications Editor

The **Color Scheme Specifications** editor includes specifications for the following characteristics:

- *Name* — The name of the color scheme.
- *Background* — The color of the background. This specification can have only one color name or value.
- *Lines* — The color names or values used for lines.
- *Text* — The color names or values used for text.

- *Polygons* The color names or values used with the `polygon`, `pie`, `barplot`, and `hist` plotting functions.

- *Images* The color names or values used with the `image` plotting function.

All color schemes must have values for the specifications **Name**, **Background**, and **Lines**. The specifications for **Text**, **Polygons**, and **Images** default to the specifications for **Lines** if left blank.

See the section Available Colors Under X11 (page 257) for information and rules on how to specify colors with the `motif` windowing graphics device.

Selecting a Different Color Scheme

To select a different color scheme, move the pointer to one of the color scheme names under the **Available Color Schemes** option menu and click. The name of the newly chosen color scheme is boxed in dashed lines, and its specifications are displayed in the **Color Scheme Specifications** editor. The plot in the graphics window, however, is still based on the original color scheme. To apply the newly chosen color scheme, you must click on the **Apply** button. (Once you apply the new color scheme, the box around the name of the new color scheme disappears.)

Figure 7.9 illustrates a setup in which there are 3 available color schemes called `color scheme 1`, `color scheme 2`, and `color scheme 3`. The default color scheme is `color scheme 1`. The specifications for this color scheme are shown in Figure 7.9 under the **Color Scheme Specifications** option menu. It uses a black background and white lines. The specifications for **Text**, **Polygons**, and **Images** are blank.

Your available color schemes will not necessarily have the names or specifications shown in Figure 7.9. (Initially, the available color schemes are defined using X resources.) How to define new color schemes and save them is explained below.

Figure 7.10 shows what happens when the color scheme `color scheme 2` is selected. Under the **Available Color Schemes** option menu, the color scheme `color scheme 2` is now boxed in dashed lines, and the specifications under the **Color Scheme Specifications** option menu have changed to the ones that correspond to `color scheme 2`.

When `color scheme 2` is applied, the example plot that you created in the section Example (page 232) has the following characteristics:

- The title, legend box, axis lines, axis labels, and axis titles are yellow (color 1).
- The points are red (color 2).
- The dashed line representing the smooth from the lowess command is cyan (color 3).

The **Available Color Schemes** option menu has enough space to show the first five available color schemes. If there are more than five available color schemes, a scrollbar appears to the right of the menu. You can view the names of the additional color schemes by using this scrollbar.

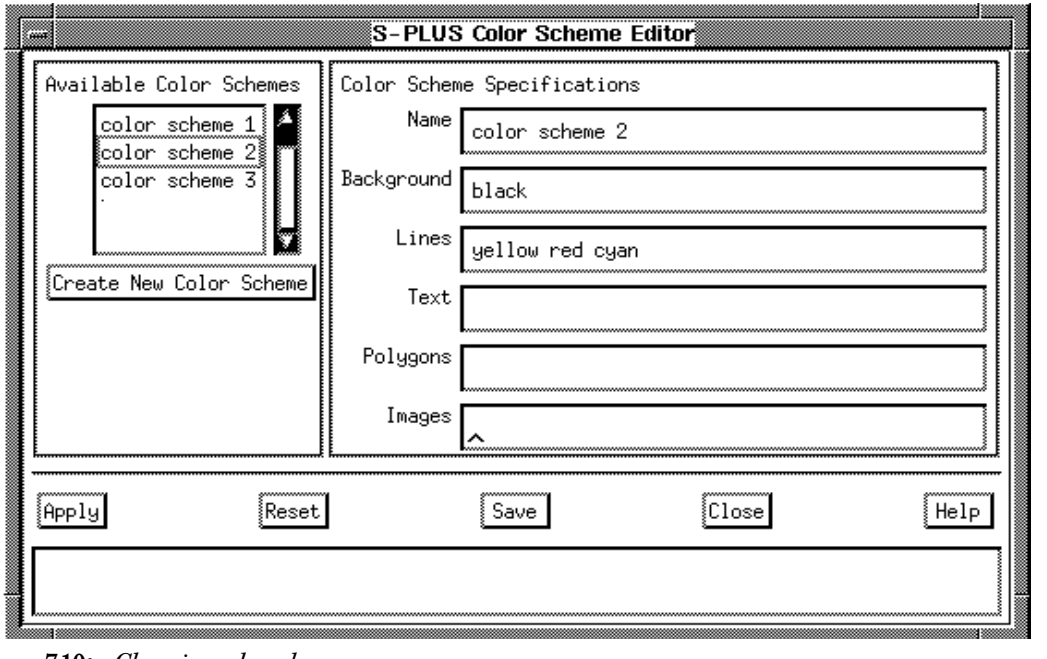


Figure 7.10: *Changing color schemes.*

Creating New Color Schemes

To create a new color scheme, follow these steps:

1. Click on the button marked **Create New Color Scheme**. Figure 7.11 shows what happens in the dialog box when you do this. The name 'unnamed' appears as the last available color scheme in the **Available Color Schemes** option menu. The default values under the **Color Scheme Specifications** option menu are the name 'unnamed', a black background, and white lines.
2. Move the pointer to the **Name** box and click. The borders of the **Name** box darken, and the cursor shape changes into an "I". Now type in text from the keyboard. To delete letters to the right of the cursor, use the DELETE key; to delete letters to the left of the cursor, use the BACKSPACE key.
3. Once you have decided on a name for the new color scheme, move the pointer to the **Background** box and follow the same procedure as in step 2. The background can only have one color value. Refer to the section Available Colors Under X11 (page 257) for information on available color names.
4. Now move the pointer to the **Lines** box and type in the desired color name(s).
5. Repeat the previous step for the **Text**, **Polygons**, and **Images** boxes.
6. To make this color scheme permanent, move the pointer to the **Save** button and click. If you do not save your newly-created color scheme, it remains only for the duration of the graphics window. Once the graphics window is destroyed, you lose any color schemes that have not been saved.
7. Move the pointer to the **Apply** button and click. The plot in the graphics window is now based on your newly-created color scheme.
8. To see the new plot, move the dialog box out of the way or click on the **Close** button to make the dialog box disappear.

The Reset Button

Any time you are in the **Color Scheme** dialog box, you may move the pointer to the **Reset** button and click. If you have not yet clicked on the **Apply** button, then the **Available Color Schemes** menu and **Color Scheme Specifications** editor are set to how they were when you first entered the dialog box. If you have at some time clicked on the **Apply** button, then the color schemes are reset to how they were immediately after the last time you clicked on the **Apply** button.

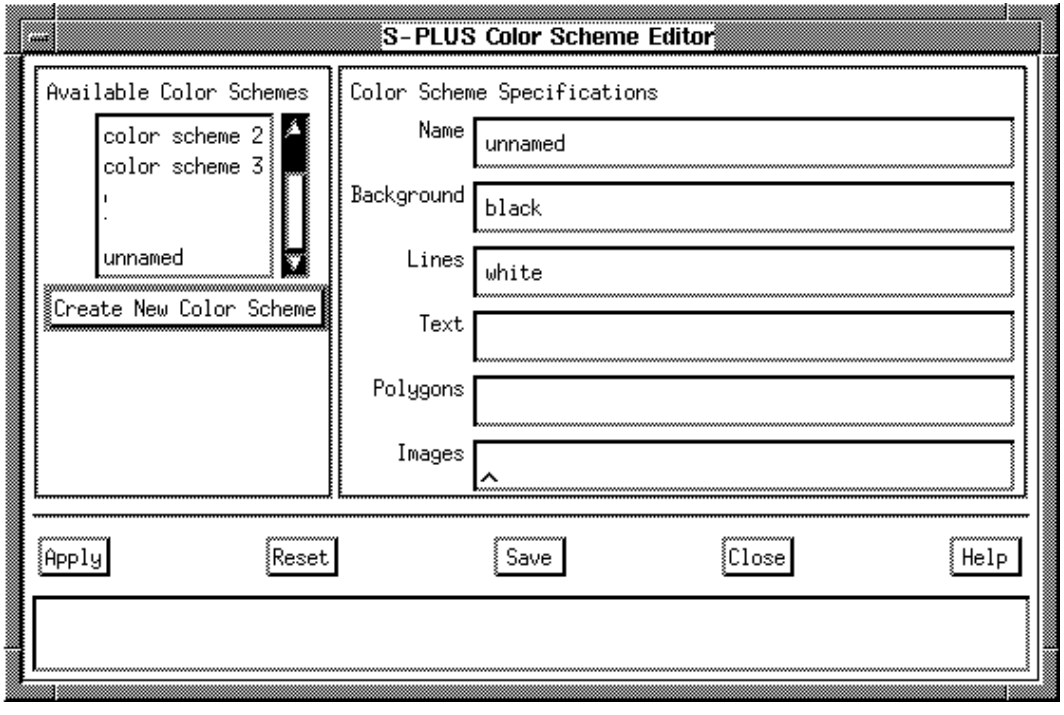


Figure 7.11: *Creating a new color scheme.*

The Printing Dialog Box

The second menu item under the **Options** menu is labeled **Printing....** When you select **Printing...**, the **Printing** dialog box appears. This window lets you interactively change the specifications of the printing method used when you choose the **Print** menu item under the **Graph** menu. (See the section The Graph Menu (page 245).)

Figure 7.12 shows an example of the **Printing** dialog box. This window has a header with a window menu button and the title S-PLUS **Graph Printing Options**. The pane of the **Printing** dialog box contains option menus entitled **Method**, **Orientation**, and (if **Method** is **LaserJet**) **Resolution**, as well as a text entry box labeled **Command**. There are also six buttons labeled **Apply**, **Reset**, **Print**, **Save**, **Close**, and **Help**. These features are explained below.

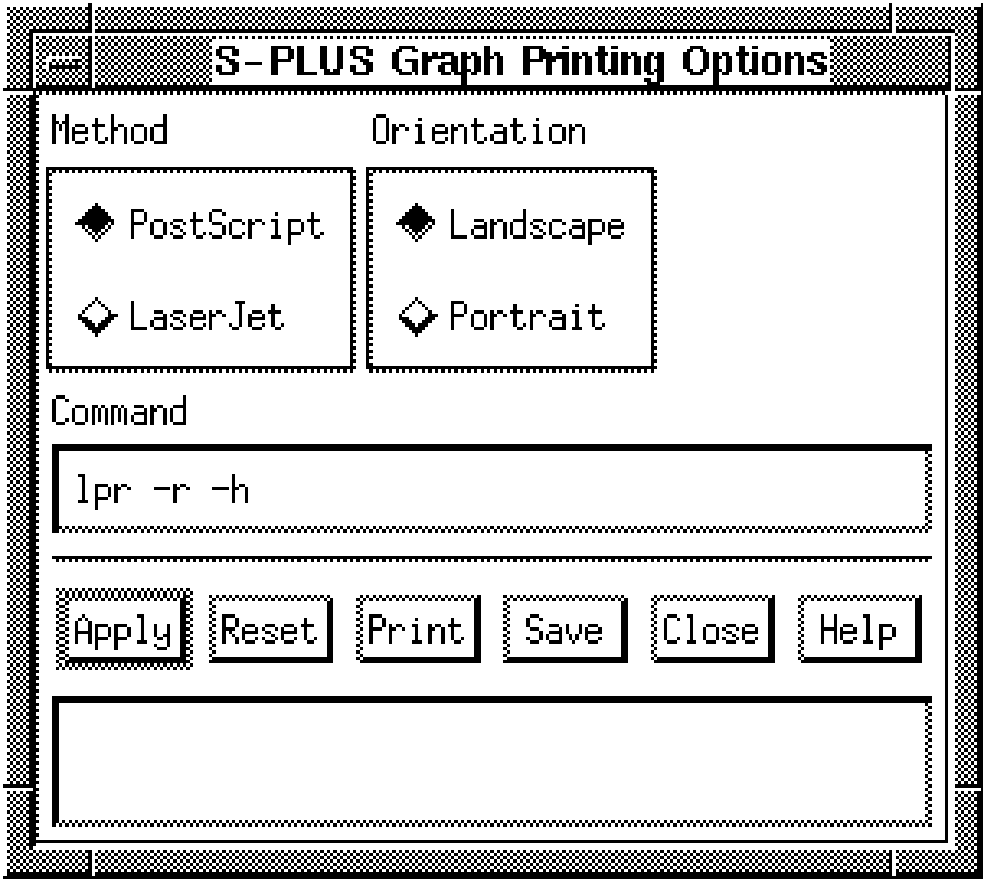


Figure 7.12: *The Motif Printing dialog box.*

Method, Orientation, Resolution, and Command

The **Method**, **Orientation**, and **Resolution** option menus all contain options marked with diamond-shaped buttons called *radio buttons*. Radio buttons are used to distinguish mutually exclusive options. The option that is currently active is denoted by a darker radio button. To change the currently active option, move the pointer to the desired option and click. These option menus and the **Command** text entry box are described below.

- *Method* — Determines the kind of file that is created when the **Print** option under the **Graph** menu is applied. The **PostScript** method produces a file of PostScript graphics commands; the **LaserJet** method produces a file of LaserJet graphics commands.

- *Orientation* — Determines the orientation of the graph on the paper. **Landscape** orientation puts the *x*-axis along the long side of the paper; **Portrait** orientation puts the *x*-axis along the short side of the paper.

- *Command* — Shows the command that is used to send the file of graphics commands to the printer. To change this command, move the pointer to this line and click. The cursor changes into an "I". You can now type in text from the keyboard.

- *Resolution* — Appears only if **Method** is set to **LaserJet**. Controls the resolution of the HP LaserJet plots.

The default settings for **Method**, **Orientation**, **Command**, and **Resolution** are initially set using X resources. The way to change these settings is explained below.

Printing Options Buttons

- *Apply* — Click on this button to apply any changes you have made to the printing specifications. Only the specifications are changed; no printing is done. Any changes you make last only as long as the graphics window remains, or until you make more changes and select **Apply** again. Once you destroy the graphics window, any changes to the original default settings are lost unless you use the **Save** button (see below).

- *Reset* — Click on this button to reset the printing specifications. If you have not yet clicked on the **Apply** button, then the specifications are set to how they were when you first entered the dialog box. If you have at some time clicked on the **Apply** button, then the specifications are reset to how they were immediately after the last time you clicked on the **Apply** button.

- *Print* — Click on this button to apply any changes you have made to the printing specifications *and* send the graph to the printer.

- *Save* — Click on this button to save the current printing specifications configuration as the default. Now every time you start S-PLUS, this configuration of default specifications appears.

- *Close* — Click on this button to make the dialog box disappear.

- *Help* — Click on this button to pop-up a **Help** window for this dialog box.

Figure 7.13 shows how the **Printing** dialog box in Figure 7.12 changes when the **Method** specification changes from **PostScript** to **LaserJet**. The **Resolution** option menu appears, and the **Command** specification for sending the graph to the printer changes.

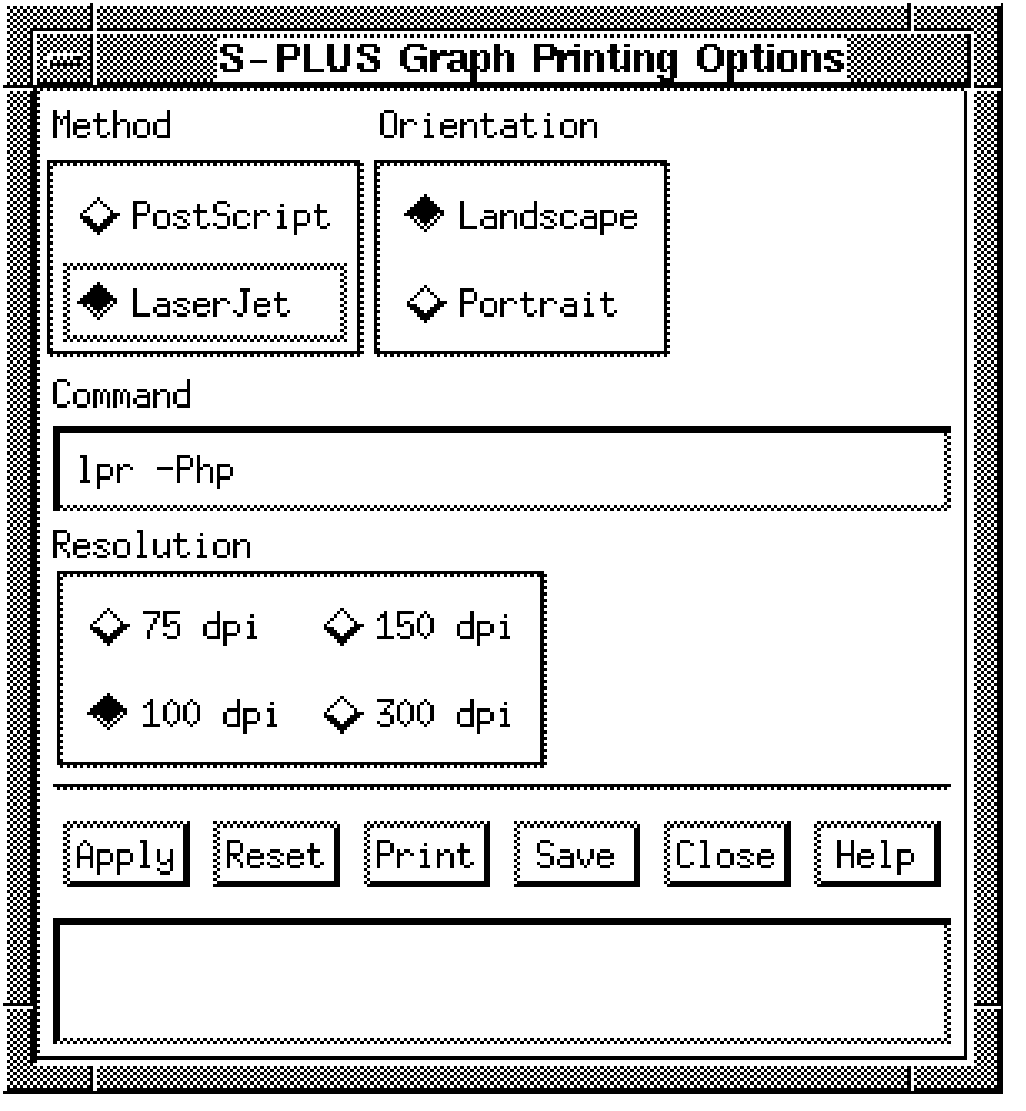


Figure 7.13: *Changing printing methods.*

Available Colors Under X11

To specify color schemes for the `motif` device, use the **Color Scheme Specifications** window.

To specify a color scheme, you must create a list of colors. There are two ways to list colors in a color scheme:

- Use color *names* listed in the system file `rgb.txt`.
- Use hexadecimal `values` that represent colors in the RGB Color Model.

The first method is a "front end" to the second method; it is easier to use, but you are limited to the colors listed in the `rgb.txt` file. The second method is more complex, but it allows you to specify any color your display is capable of producing. Both methods are described below.

The initial set of colors is set system-wide at installation. Any changes you make using the **Color Scheme Specifications** window override the system values. This remains true even if system-wide changes are installed.

Viewing Color Names Listed in rgb.txt

The **rgb.txt** file contains a list of predefined colors that have been translated from a hexadecimal code into English text. To see what the available color names are, you can either look at the **rgb.txt** file with a text editor, or you can use the **showrgb** command coupled with a paging program like **more** by typing the following command:

```
showrgb | more
```

The `rgb.txt` file is usually located in the directory **/usr/lib/X11** . To move into this directory, type the command

```
cd /usr/lib/X11
```

Table 7.1 gives some examples of available colors in the **rgt.txt** file.

Table 7.1: *Some available colors in* `rgb.txt`.

violet	blue	green	yellow
orange	red	black	white
ghost white	peach puff	lavender blush	lemon chiffon
lawn green	chartreuse	olive drab	lime green
magenta	medium orchid	blue violet	purple

Hexadecimal Color Values

You can also specify a color by using a hexadecimal value from the Red, Green, and Blue (RGB) Color Model. (A hexadecimal value is made up of hexadecimal digits. A hexadecimal digit can take on any of the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, listed from smallest to largest.) Most color displays are based on the RGB Color Model. Each pixel on the screen is made up of three phosphors: one red, one green, and one blue. Varying the intensities of each of these phosphors varies the color that you see on your display.

You can specify the intensities of each of the three phosphors with a hexadecimal triad. The first part of the triad corresponds to the intensity of the red phosphor, the second to the intensity of the green phosphor, and the third to the intensity of the blue phosphor. A hexadecimal triad must begin with the symbol #. For example, the hexadecimal triad `#000` corresponds to no intensity in any of the phosphors and yields the color black, while the triad `#FFF` corresponds to maximum intensity in all of the phosphors and yields white.

A hexadecimal triad with only one digit per phosphor allows for 4,096 (16^3) colors. Most displays are capable of many more colors than this, so you can use more than one digit per phosphor. Table 7.2 shows the allowed forms for an RGB triad; Table 7.3 illustrates hexadecimal values for some common colors. You can use up to four digits to specify the intensity of one phosphor (this allows for about 3 x $\$10^{14}$ colors). You do not need to know how many colors your machine can display; your window system automatically scales the color specifications to your hardware.

Table 7.2: *Legal forms of RGB triads.*

Triad Form	Approximate Number of Possible Colors
#RGB	4,000
#RRGGBB	17 million
#RRRGGGBBB	70 billion
#RRRRGGGGBBBB	3×10^{14}

Table 7.3: *Hexadecimal values of some common colors.*

Hex Value	Color Name
#000000	black
#FFFFFF	white
#FF0000	red
#00FF00	green
#0000FF	blue
#FFFF00	yellow
#00FFFF	cyan
#FF00FF	magenta
#ADD8E6	light blue

Specifying Color Schemes

The following conventions are used when listing colors to specify a color scheme:

- Color names or values are separated by spaces.
- When a color name is more than one word, it should be enclosed in quotes. For example, "lawn green".
- The order in which you list the color names or values corresponds to the numerical order in which they are referred to in S-PLUS with the graphics parameter `col`. For example, if you use the argument `col=3` in an S-PLUS plotting function, you are referring to the third color listed in the current color scheme.

Note

When specifying a color scheme in your X resources, the first color listed is the background color and corresponds to `col=0`.

- Colors are repeated cyclically, starting with color 1 (which corresponds to `col=1`). For example, if the current color scheme includes three colors (not including the background color), and you use the argument `col=5` in an S-PLUS plotting function, then the second color is used.
- You may abbreviate a list of colors with the specification *color1 n color2*. This list is composed of (*n*+2) colors: *color1*, *color2*, and *n* colors that range smoothly between *color1* and *color2*. For example, the color scheme `blue red 10 "lawn green"` specifies a list of 13 colors: blue, then red, then 10 colors ranging in between red and lawn green, and then lawn green.

Note

This method of specification is especially useful with the `image` plotting function.

- You may specify a list of colors as *halftones* with the specification *color1 hn color2*. This list is composed of (*n*+2) "colors," which are actually tile patterns with progressively more *color2* on a background of *color1*. Halftone specifications are useful on devices with a limited number of simultaneous colors. For example, the color scheme `blue red h10 "lawn green"` specifies a list of 13 colors, just as our previous example did. In this example, however, only 3 entries in the X server's color table are allocated, rather than the 13 allocated by the previous example.

STATISTICS 8

Introduction **264**
- Overview 264
- Basic Procedure 266
- Dialogs 266
- Dialog Fields 267
- Plotting From the Statistics Dialogs 268
- Statistics Options 268
- Saving Results From an Analysis 268

Summary Statistics **269**
- Summary Statistics 269
- Crosstabulations 271
- Correlations 274

Compare Samples **276**
- One-Sample Tests 276
- Two-Sample Tests 287
- K-Sample Tests 298
- Counts and Proportions 308

Power and Sample Size **322**
- Normal Mean 322
- Binomial Proportion 324

Experimental Design **327**
- Factorial 327
- Orthogonal Array 328
- Design Plot 329
- Factor Plot 330
- Interaction Plot 332

Regression **334**
- Linear Regression 335
- Robust MM Regression 341
- Robust LTS Regression 343

Stepwise Linear Regression 344
Generalized Additive Models 347
Local (Loess) Regression 348
Nonlinear Regression 349
Generalized Linear Models 354
Log-Linear (Poisson) Regression 355
Logistic Regression 356
Probit Regression 359

Analysis of Variance 361
Fixed Effects ANOVA 361
Random Effects ANOVA 362
Multiple Comparisons 364

Mixed Effects 367
Linear 367
Nonlinear 368

Generalized Least Squares 371
Linear 371
Nonlinear 372

Survival 375
Nonparametric Survival 375
Cox Proportional Hazards 376
Parametric Survival 378
Life Testing 379

Tree 381
Tree Models 381
Tree Tools 382

Compare Models 386

Cluster Analysis 389
Compute Dissimilarities 389
K-Means Clustering 390
Partitioning Around Medoids 392
Fuzzy Partitioning 393
Agglomerative Hierarchical Clustering 395
Divisive Hierarchical Clustering 397
Monothetic Clustering 399

Multivariate **401**
Discriminant Analysis 401
Factor Analysis 402
Principal Components 404
MANOVA 406

Quality Control Charts **408**
Continuous Grouped 408
Continuous Ungrouped 409
Counts and Proportions 411

Resample **413**
Bootstrap Inference 413
Jackknife Inference 415

Smoothing **417**
Kernel Smoother 418
Local Regression (Loess) 418
Spline Smoother 419
Supersmoother 419
Examples 420

Time Series **421**
Autocorrelations 421
ARIMA 424
Lag Plot 426
Spectrum Plot 427

References **428**

INTRODUCTION

The power of S-PLUS comes from the integration of its graphics capabilities with its statistical analysis routines. In other chapters throughout this manual, we introduce S-PLUS graphics. In this chapter, we show how statistical procedures are performed in S-PLUS.

It is not necessary to read this entire chapter before you perform a statistical analysis. Once you've acquired a basic understanding of the way statistics are performed, you can refer directly to a section of interest.

We begin this chapter by presenting general information on using the statistics dialogs, and devote the remaining sections to descriptions and examples for each of these dialogs. Wherever possible, we complement statistical examples with plots generated by the graphics dialogs. However, not all of the S-PLUS functionality has been built into the menu options, and it is therefore necessary to use command line functions in some sections.

Overview

Figure 8.1 displays many elements of the S-PLUS interface.

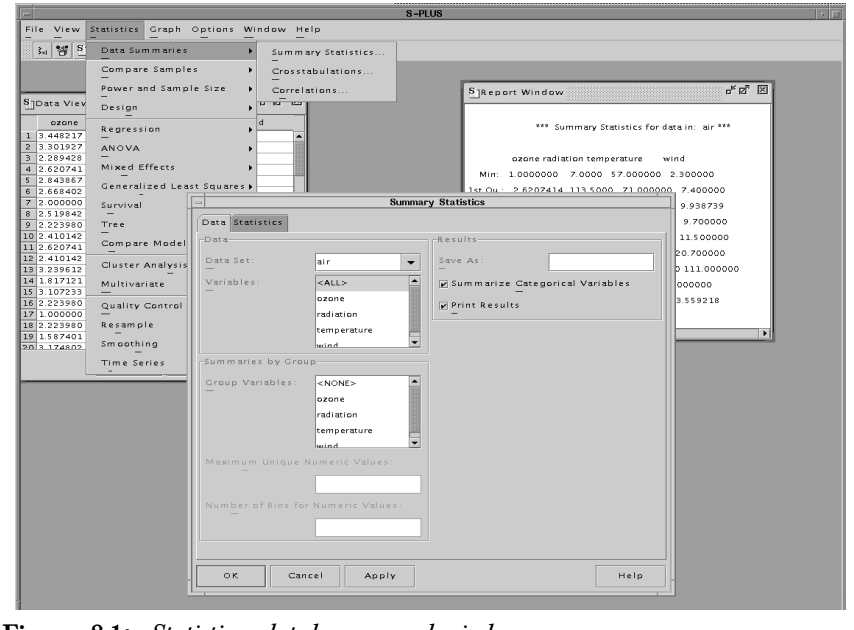


Figure 8.1: *Statistics-related menus and windows.*

- **Statistics menu**: The **Statistics** menu gives you access to nearly all of the statistical procedures available in S-PLUS. The procedures are logically grouped, with submenus that allow you to precisely specify the procedure you want to use. For example, in Figure 8.1 the menu tree for summary statistics is shown. It is selected by choosing **Statistics ► Data Summaries ► Summary Statistics**.

- **Statistics dialogs**: The open dialog in Figure 8.1 is entitled **Summary Statistics**, and is used to specify which data summaries to calculate.

- **Data Viewer**: The open window on the left in Figure 8.1 is a **Data** viewer, which you can use to see a data set in its entirety. The **Data** viewer is not a data editor, however, and you cannot use it to modify or create a new data set.

- **Report Window**: The **Report** window displays the results of statistical analyses. In Figure 8.1, a **Report** window shows the results of the chosen summary statistics. In addition, any error, warning, or informational message generated by a statistics dialog is printed in the **Report** window.

- **Commands Window** (not shown): The **Commands** window contains the S-PLUS command line prompt, which you can use to call S-PLUS functions that are not yet implemented in the menu options.

- **Graph Window** (not shown): A **Graph** window displays the graphics created from the statistics menus.

Basic Procedure

The basic procedure for analyzing data is the same regardless of the type of analysis.

1. Choose the statistical procedure (summary statistics, linear regression, ANOVA, etc.) you want to perform from the **Statistics** menu. The dialog corresponding to that procedure opens.
2. Select the data set, variables, and options for the procedure you have chosen. (These are slightly different for each dialog.) Click the **OK** or **Apply** button to conduct the analysis. If you click **OK**, the dialog closes when the graph is generated; if you click **Apply**, the dialog remains open.
3. Check for messages. If a message is generated, it appears in the **Report** window.
4. Check the result. If everything went well, the results of your analysis are displayed in the **Report** window. Some statistics procedures also generate plots.

If you want, you can change the variables, parameters, or options in the dialog and click **Apply** to generate new results. S-PLUS makes it easy to experiment with options and to try variations on your analysis.

Dialogs

Most of the statistical functionality of S-PLUS can be accessed through the **Statistics** menus. The **Statistics** menu includes dialogs for creating data summaries and fitting statistical models. Many of the dialogs consist of tabbed pages that allow for a complete analysis, including model fitting, plotting, and prediction. Each dialog has a corresponding function that is executed using dialog inputs as values for function arguments. Usually, it is only necessary to fill in a few fields on the first page of a tabbed dialog to launch the function call.

Dialog Fields

Many dialogs include a **Data Set** field. To specify a data set, you can either type its name directly in the **Data Set** field, or make a selection from the dropdown list. Note that the **Data Set** field recognizes objects of class `"data.frame"` only, and does not accept matrices, vectors, or time series. For this reason, we periodically drop to the **Commands** window in this chapter to create objects that are accepted by the menu options.

Most dialogs that fit statistical models include a **Subset Rows** field that you can use to specify only a portion of a data set. To use a subset of your data in an analysis, enter an S-PLUS expression in the **Subset Rows** field that identifies the rows to use. The expression can evaluate to a vector of logical values: true values indicate which rows to include in the analysis, and false values indicate which rows to drop. Alternatively, the expression can specify a vector of row indices. For example:

- The expression `Species=="bear"` includes only rows for which the `Species` column contains `bear`.
- The expression `Age>=13 & Age<20` includes only rows that correspond to teenage values of the `Age` variable.
- The expression `1:20` includes the first 20 rows of the data.

To use all rows in a data set, leave the **Subset Rows** field blank.

Some dialogs require a **Formula**. To specify a formula, you can type one directly in the **Formula** field, or click the **Create Formula** button to bring up a dialog that builds a formula for you. Some dialogs, such as the **Generalized Additive Models** dialog, require special formulas; in these cases, the special terms available are listed in the **Formula Builder**.

Most dialogs have a **Save As** field that corresponds to the name of the object in which the results of the analysis are saved. Many of the modeling dialogs also have one or more **Save In** fields. The **Save In** field corresponds to the name of a data set in which new columns are saved. Examples of new columns include fitted values, residuals, predictions, and standard errors.

Plotting From the Statistics Dialogs

Most of the statistics dialogs produce default plots that are appropriate for the analysis. Many have several plot options, usually on a separate **Plot** tab.

The **Options** menu contains a few options that affect the graphics you create from the statistics menus. In particular:

- The **Options ► Dialog Options** window includes a **Create New Graph Window** check box. If this box is selected, as it is by default, then a new **Graph** window is created each time you generate a statistics plot.
- The **Options ► Set Graph Colors** window allows you to select a color scheme for your graphics.
- The **Options ► Graph Options** window governs whether tabbed pages in **Graph** windows are deleted, preserved, or written over when a new plot is generated.

Statistics Options

The **Options ► Dialog Options** window includes an **Echo Dialog Command** check box. If this box is selected, the command associated with a dialog action is printed before its output in the **Report** window. This allows you to copy and paste the commands used for your analyses into your own S-PLUS functions.

Saving Results From an Analysis

A statistical model object may be created by specifying a name for the object in the **Save As** field of a dialog. Once the execution of a dialog function completes, the object shows up in your working database. You can then access the object from the **Commands** window. This allows you to do plotting and prediction for a model without relaunching an entire dialog.

SUMMARY STATISTICS

One of the first steps in analyzing data is to create summaries. This can be done numerically through the **Summary Statistics**, **Crosstabulations**, and **Correlations and Covariances** dialogs.

- **Summary Statistics:** calculates summary statistics, such as the mean, median, variance, total sum, quartiles, etc.
- **Crosstabulations:** tabulates the number of cases for each combination of factors between your variables, and generates statistics for the table.
- **Correlations:** calculates correlations or covariances between variables.

These three procedures can be found under the **Statistics ► Data Summaries** menu.

Summary Statistics

The **Summary Statistics** dialog provides basic univariate summaries for continuous variables, and it provides counts for categorical variables. Summaries may be calculated within groups based on one or more grouping variables.

Computing summary statistics

From the main menu, choose **Statistics ► Data Summaries ► Summary Statistics**. The **Summary Statistics** dialog opens, as shown in Figure 8.2.

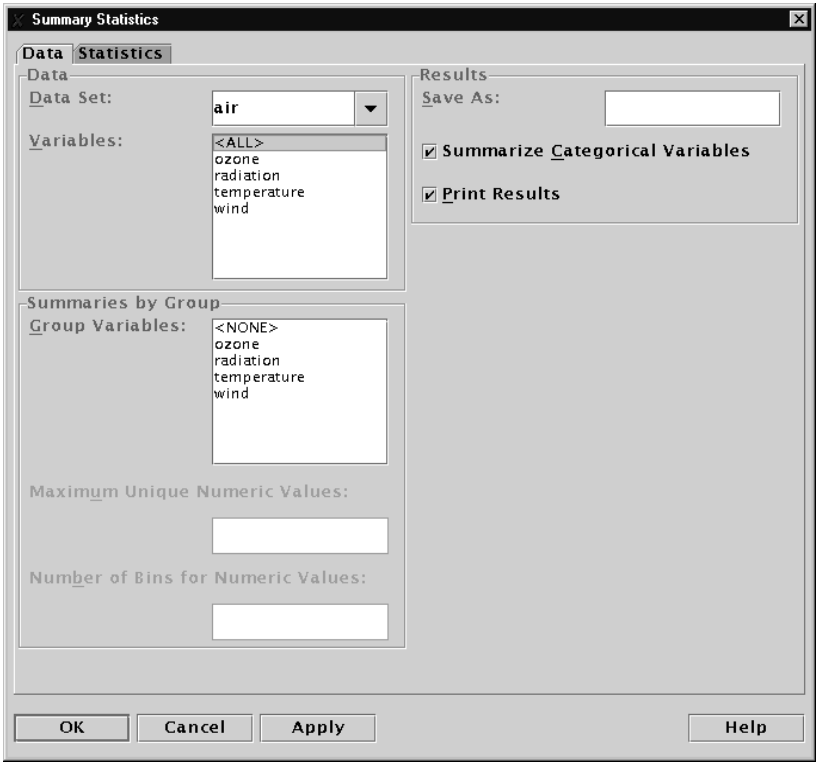


Figure 8.2: *The **Summary Statistics** dialog.*

Example

We use the data set `air`. This data set measures the ozone concentration, wind speed, temperature, and radiation of 111 consecutive days in New York. In this example, we calculate summary statistics for these data.

1. Open the **Summary Statistics** dialog.
2. Type `air` in the **Data Set** field.
3. Select the variables you want summary statistics for in the **Variables** field. For this example, we choose **<ALL>** (the default).

4. Click on the **Statistics** tab to see the statistics available. For this example, select the **Variance** and **Total Sum** check boxes.
5. Make sure the **Print Results** check box is selected to ensure that the results are printed in the **Report** window.
6. Click **OK**. A **Report** window containing the following output is created, if one does not already exist:

```
*** Summary Statistics for data in:  air ***

             ozone radiation temperature    wind
Min:          1.00      7.00       57.00    2.30
1st Qu.:      2.62    113.50       71.00    7.40
Mean:         3.25    184.80       77.79    9.94
Median:       3.14    207.00       79.00    9.70
3rd Qu.:      3.96    255.50       84.50   11.50
Max:          5.52    334.00       97.00   20.70
Total N:    111.00    111.00      111.00  111.00
NA's :        0.00      0.00        0.00    0.00
Variance:     0.79   8308.74       90.82   12.67
Std Dev.:     0.89     91.15        9.53    3.56
Sum:        360.50  20513.00     8635.00 1103.20
```

7. If the above output is not displayed, check the **Report** window for error messages.

We are done. As you can see, calculating summary statistics is straightforward. Other statistical procedures use the same basic steps that we did in this example.

Crosstabulations

The **Crosstabulations** dialog produces a table of counts for all combinations of specified categorical (factor) variables. In addition, it calculates cell percentages and performs a chi-square test for independence. The **Crosstabulations** dialog returns results in an ASCII formatted table.

The chi-square test for independence is useful when the data consist of the number of occurrences of an outcome for various combinations of categorical covariates. It is used to determine whether the number of occurrences is due to the marginal values of the covariates, or whether it is influenced by an interaction between covariates.

Computing crosstabulations

From the main menu, choose **Statistics ▶ Data Summaries ▶ Crosstabulations**. The **Crosstabulations** dialog opens, as shown in Figure 8.3.

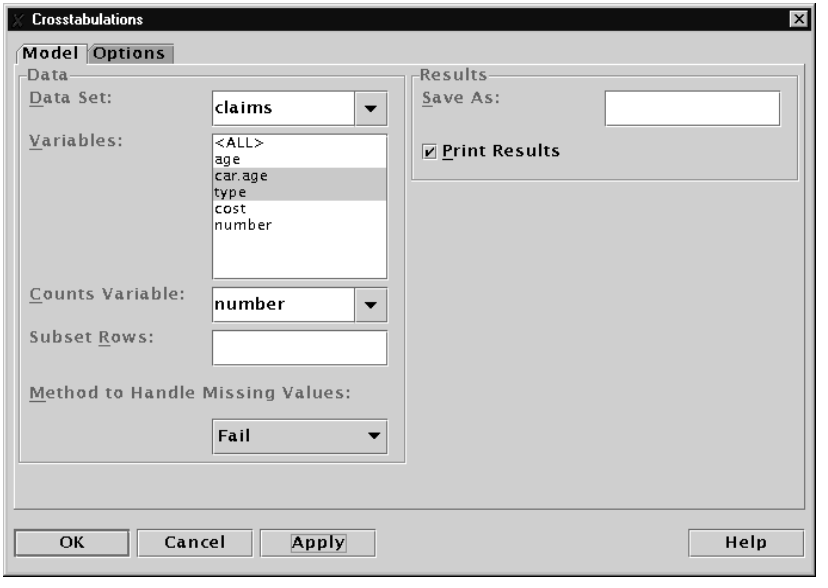


Figure 8.3: *The **Crosstabulations** dialog.*

Example

Consider the data set `claims`, which has the components `age`, `car.age`, `type`, `cost`, and `number`. The original data were taken from 8,942 insurance claims. The 128 rows of the `claims` data set represent all possible combinations of the three predictor variables (columns) `age`, `car.age`, and `type`. An additional variable, `number`, gives the number of claims in each cell. The outcome variable, `cost`, is the average cost of the claims.

We can use a contingency table to examine the distribution of the number of claims by car age and type. The corresponding test for independence tells us whether the effect of age upon the likelihood of a claim occurring varies by car type, or whether the effects of car age and type are independent.

To construct a contingency table for the `claims` data:

1. Open the **Crosstabulations** dialog.
2. Type `claims` in the **Data Set** field.
3. In the **Variables** field, click on `car.age` and then CTRL-click `type`. This selects both variables for the analysis.
4. In the **Counts Variable** field, scroll through the list of variables and select `number`.
5. Click **OK**.

The table below appears in the **Report** window. Each cell in the table contains the number of claims for that car age and type combination, along with the row percentage, column percentage, and total percentage of observations falling in that cell. The results of the test for independence indicate that the percentage of observations in each cell is significantly different from the product of the total row percentage and total column percentage. Thus, there is an interaction between the car age and type, which influences the number of claims. That is, the effect of car age on the number of claims varies by car type.

```
Call:
crosstabs(formula = number ~ car.age + type, data =
  claims, na.action = na.fail, drop.unused.levels = T)
8942 cases in table
+----------+
|N         |
|N/RowTotal|
|N/ColTotal|
|N/Total   |
+----------+
car.age|type
       |A      |B      |C      |D      |RowTotl|
-------+-------+-------+-------+-------+-------+
0-3    | 391   |1538   |1517   | 688   |4134   |
       |0.0946 |0.3720 |0.3670 |0.1664 |0.462  |
       |0.3081 |0.3956 |0.5598 |0.6400 |       |
       |0.0437 |0.1720 |0.1696 |0.0769 |       |
-------+-------+-------+-------+-------+-------+
```

```
4-7     | 538   |1746   | 941   | 324   |3549   |
        |0.1516 |0.4920 |0.2651 |0.0913 |0.397  |
        |0.4240 |0.4491 |0.3472 |0.3014 |       |
        |0.0602 |0.1953 |0.1052 |0.0362 |       |
--------+-------+-------+-------+-------+-------+
8-9     | 187   | 400   | 191   |  44   |822    |
        |0.2275 |0.4866 |0.2324 |0.0535 |0.092  |
        |0.1474 |0.1029 |0.0705 |0.0409 |       |
        |0.0209 |0.0447 |0.0214 |0.0049 |       |
--------+-------+-------+-------+-------+-------+
10+     | 153   | 204   |  61   |  19   |437    |
        |0.3501 |0.4668 |0.1396 |0.0435 |0.049  |
        |0.1206 |0.0525 |0.0225 |0.0177 |       |
        |0.0171 |0.0228 |0.0068 |0.0021 |       |
--------+-------+-------+-------+-------+-------+
ColTotl|1269   |3888   |2710   |1075   |8942   |
        |0.14   |0.43   |0.30   |0.12   |       |
--------+-------+-------+-------+-------+-------+
Test for independence of all factors
  Chi^2 = 588.2952 d.f.= 9 (p=0)
  Yates' correction not used
```

Correlations

The **Correlations and Covariances** dialog produces the basic bivariate summaries of correlations and covariances.

Computing correlations and covariances

From the main menu, choose **Statistics** ▶ **Data Summaries** ▶ **Correlations**. The **Correlations and Covariances** dialog opens, as shown in Figure 8.4.

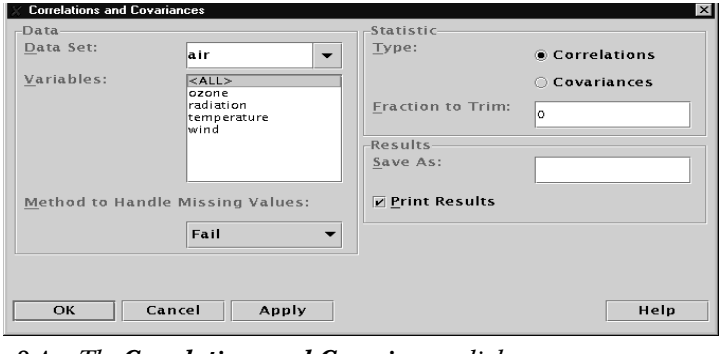


Figure 8.4: *The **Correlations and Covariances** dialog.*

Example

In the section Summary Statistics on page 269, we looked at univariate summaries of the data set `air`. We now generate the correlations between all four variables of the data set. Here are the basic steps:

1. Open the **Correlations and Covariances** dialog.
2. Type `air` in the **Data Set** field.
3. Choose **<ALL>** in the **Variables** field.
4. Click **OK**.

The **Report** window displays the correlations between the four variables:

```
*** Correlation for data in: air ***

                ozone  radiation temperature       wind
ozone       1.0000000  0.4220130   0.7531038 -0.5989278
radiation   0.4220130  1.0000000   0.2940876 -0.1273656
temperature 0.7531038  0.2940876   1.0000000 -0.4971459
wind       -0.5989278 -0.1273656  -0.4971459  1.0000000
```

Note the strong correlation of 0.75 between `ozone` and `temperature`: as temperature increases, so do the ozone readings. The negative correlation of -0.60 between `ozone` and `wind` indicates that ozone readings decrease as the wind speed increases. Finally, the correlation of -0.50 between `wind` and `temperature` indicates that the temperature decreases as the wind increases (or that the temperature increases as the wind decreases).

COMPARE SAMPLES

One-Sample Tests

S-PLUS supports a variety of statistical tests for testing a hypothesis about a single population. Most of these tests involve testing a parameter against a hypothesized value. That is, the null hypothesis has the form H_0: $\Theta = \Theta_0$, where Θ is the parameter of interest and Θ_0 is the hypothesized value of our parameter.

- **One-sample t-test:** a test for the population mean μ. We test if the population mean is a certain value. For small data sets, we require that the population have a normal distribution.
- **One-sample Wilcoxon signed-rank test:** a nonparametric test for the population mean μ. As with the t-test, we test if the population mean is a certain value, but we make no distributional assumptions.
- **One-sample Kolmogorov-Smirnov goodness-of-fit test:** a test to determine if the data come from a hypothesized distribution. This is the preferred goodness-of-fit test for a continuous variable.
- **One-sample chi-square goodness-of-fit test:** a test to see if the data come from a hypothesized distribution. This is the preferred goodness-of-fit test for a discrete variable.

One-Sample t-Test

A *one-sample t-test* is used to test whether the mean for a variable has a particular value. The main assumption in a t-test is that the data come from a Gaussian (normal) distribution. If this is not the case, then a nonparametric test, such as the Wilcoxon signed-rank test, may be a more appropriate test of location.

Performing a one-sample t-test

From the main menu, choose **Statistics ► Compare Samples ► One Sample ► t Test**. The **One-sample t Test** dialog opens, as shown in Figure 8.5.

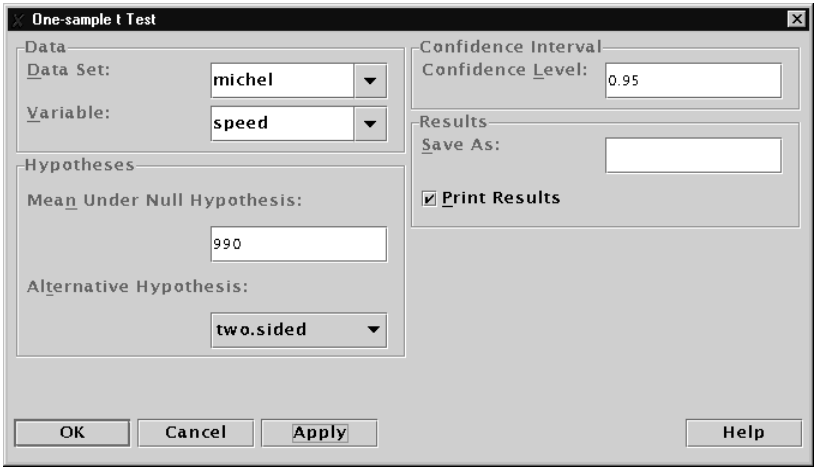


Figure 8.5: *The **One-sample t Test** dialog.*

Example

In 1876, the French physicist Cornu reported a value of 299,990 km/sec for c, the speed of light. In 1879, the American physicist A.A. Michelson carried out several experiments to verify and improve Cornu's value. Michelson obtained the following 20 measurements of the speed of light:

```
 850  740  900 1070  930  850  950  980  980  880
1000  980  930  650  760  810 1000 1000  960  960
```

To obtain Michelson's actual measurements, add 299,000 km/sec to each of the above values.

In the chapter Menu Graphics, we created a `michel` data set containing the Michelson data. For convenience, we repeat the S-PLUS command here:

```
> michel <- data.frame(speed = c(850, 740, 900,
+ 1070, 930, 850, 950, 980, 980, 880, 1000, 980,
+ 930, 650, 760, 810, 1000, 1000, 960, 960))
```

Exploratory data analysis

To obtain a useful exploratory view of the Michelson data, create the following plots: a boxplot, a histogram, a density plot, and a QQ normal plot. You can create these plots from the **Graph** menu or the **Commands** window. The function below packages the four exploratory data analysis (EDA) plots into one S-PLUS call:

```
> eda.shape <- function(x) {
+     par(mfrow = c(2, 2))
+     hist(x)
+     boxplot(x)
+     iqd <- summary(x)[5] - summary(x)[2]
+     plot(density(x, width = 2 * iqd),
+         xlab = "x", ylab = "", type = "l")
+     qqnorm(x)
+     qqline(x)
+     invisible()
+ }

> eda.shape(michel$speed)
```

The plots that `eda.shape` generates for the Michelson data are shown in Figure 8.6. We want to evaluate the shape of the distribution to see if our data are normally distributed. These plots reveal a distinctly skewed distribution toward the left (that is, toward smaller values). The distribution is thus not normal and probably not even "nearly" normal. We should therefore not use Student's t-test for our statistical inference, since it requires normality for small samples.

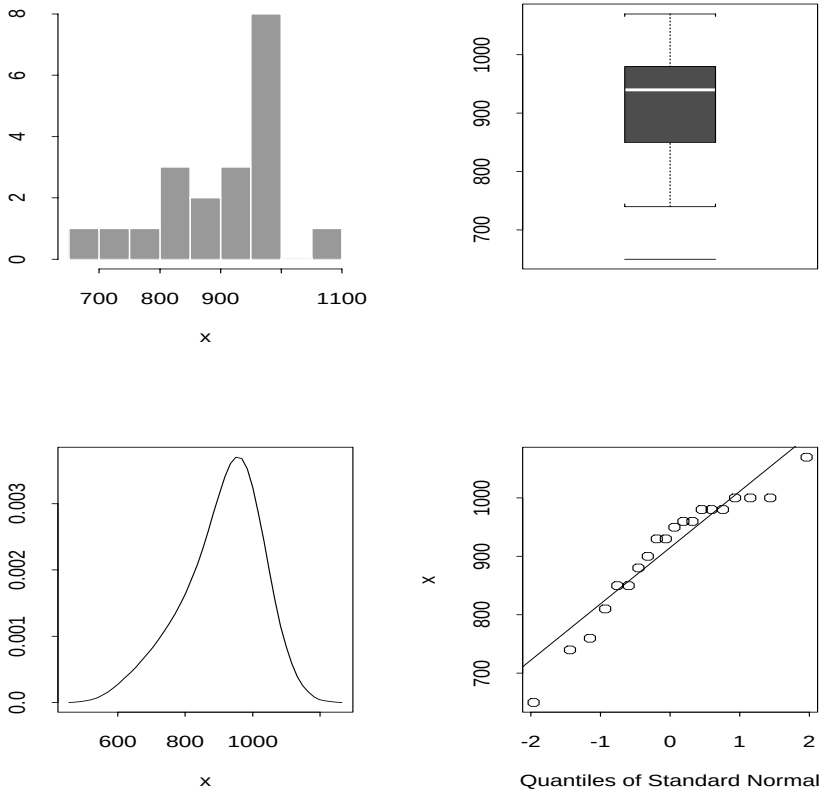


Figure 8.6: *Exploratory data analysis plots for the Michelson data.*

The solid horizontal line in the box plot is located at the *median* of the data, and the upper and lower ends of the box are located at the *upper* and *lower quartiles* of the data, respectively. To obtain precise values for the median and quartiles, use the **Summary Statistics** dialog.

1. Open the **Summary Statistics** dialog.
2. Enter `michel` as the **Data Set**.

3. Click on the Statistics tab, and deselect all options except **Mean**, **Minimum**, **First Quartile**, **Median**, **Third Quartile**, and **Maximum**.
4. Click **OK**. The output appears in the **Report** window.

```
***  Summary Statistics for data in:  michel ***

    Min:  650.000
1st Qu.:  850.000
   Mean:  909.000
 Median:  940.000
3rd Qu.:  980.000
    Max: 1070.000
```

The summary shows, from top to bottom, the smallest observation, the first quartile, the mean, the median, the third quartile, and the largest observation. From this summary, you can compute the *interquartile range*, $IQR = 3Q - 1Q$. The interquartile range provides a useful criterion for identifying outliers: any observation that is more than 1.5 3 *IQR* above the third quartile or below the first quartile is a suspected outlier.

Statistical inference

Because the Michelson data are probably not normal, you should use the Wilcoxon signed-rank test for statistical inference, rather than the Student's t-test. For illustrative purposes, we use both.

To compute Student's t confidence intervals for the population mean-value location parameter μ, we use the **One-sample t Test** dialog. This dialog also computes Student's t significance test p-values for the parameter $\mu_0 = 299,990$.

1. Open the **One-sample t Test** dialog.
2. Type `michel` in the **Data Set** field.
3. Select `speed` as the **Variable**.

4. Suppose you want to test the null hypothesis value $\mu_0 = 990$ (plus 299,000) against a two-sided alternative, and you want to construct 95% confidence intervals. Enter 990 as the **Mean Under Null Hypothesis**.
5. Click **OK**.

The results of the one-sample t-test appear in the **Report** window.

```
One-sample t-Test

data:  speed in michel
t = -3.4524, df = 19, p-value = 0.0027
alternative hypothesis: true mean is not equal to 990
95 percent confidence interval:
 859.8931 958.1069
sample estimates:
 mean of x
       909
```

The computed mean of the Michelson data is 909, and the p-value is 0.0027, which is highly significant. Clearly, Michelson's average value of 299,909 km/sec for the speed of light is significantly different from Cornu's value of 299,990 km/sec.

S-PLUS returns other useful information besides the p-value, including the t-statistic value, the degrees of freedom, the sample mean, and the confidence interval.

One-Sample Wilcoxon Signed-Rank Test

The *Wilcoxon signed-rank test* is used to test whether the median for a variable has a particular value. Unlike the one-sample t-test, it does not assume that the observations come from a Gaussian (normal) distribution.

Performing a one-sample Wilcoxon signed-rank test

From the main menu, choose **Statistics ▶ Compare Samples ▶ One Sample ▶ Wilcoxon Signed-Rank Test**. The **One-sample Wilcoxon Test** dialog opens, as shown in Figure 8.7.

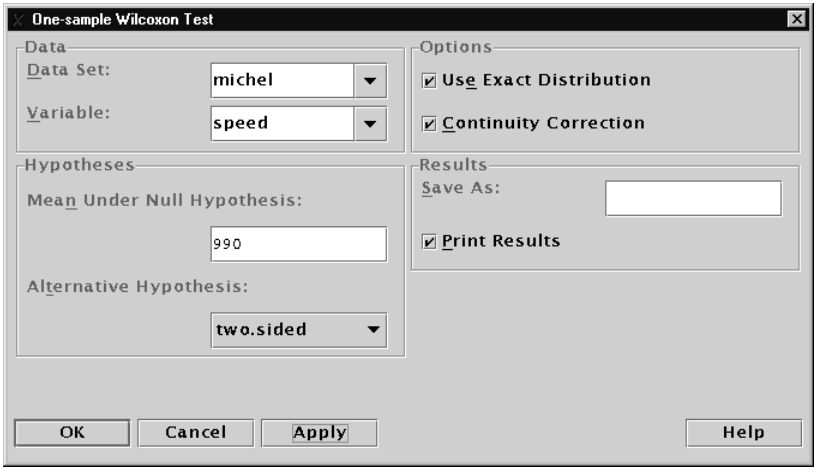


Figure 8.7: *The **One-sample Wilcoxon Test** dialog.*

Example

In the section One-Sample t-Test on page 276, we performed a t-test on the Michelson data. The test concludes that Michelson's average value for the speed of light (299,909 km/sec) is significantly different from Cornu's value of 299,990 km/sec. However, we have noted that the data may not be normal, so the results of the t-test are suspect. We now conduct a Wilcoxon signed-rank test to see if the two values for the speed of light differ significantly from each other.

1. If you have not done so already, create the `michel` data set with the instructions given on page 160 in the Menu Graphics chapter.
2. Open the **One-sample Wilcoxon Test** dialog.
3. Type `michel` in the **Data Set** field.
4. Select `speed` as the **Variable**.
5. Enter 990 as the **Mean Under Null Hypothesis**.
6. Click **OK**.

The **Report** window shows:

```
        Wilcoxon signed-rank test

data:  speed in michel
signed-rank normal statistic with correction Z = -3.0715,
  p-value = 0.0021
alternative hypothesis: true mu is not equal to 990
```

You may also receive a warning message that there are duplicate values in the variable `speed`. You can ignore this message. The p-value of 0.0021 is close to the t-test p-value of 0.0027 for testing the same null hypothesis with a two-sided alternative. Thus, the Wilcoxon signed-rank test confirms that Michelson's average value for the speed of light of 299,909 km/sec is significantly different from Cornu's value of 299,990 km/sec.

Kolmogorov-Smirnov Goodness-of-Fit

The *Kolmogorov-Smirnov goodness-of-fit test* is used to test whether the empirical distribution of a set of observations is consistent with a random sample drawn from a specific theoretical distribution. It is generally more powerful than the chi-square goodness-of-fit test for continuous variables. For discrete variables, the chi-square test is generally preferable.

If parameter values for the theoretical distribution are not available, they may be estimated from the observations automatically as part of the test for normal (Gaussian) or exponential distributions. For other distributions, the chi-square test must be used if parameters are to be estimated. In this case, the parameters are estimated from the data separately from the test, and then entered into the dialog.

Performing a one-sample Kolmogorov-Smirnov goodness-of-fit test

From the main menu, choose **Statistics ▶ Compare Samples ▶ One Sample ▶ Kolmogorov-Smirnov GOF**. The **One-sample Kolmogorov-Smirnov Goodness-of-Fit Test** dialog opens, as shown in Figure 8.8.

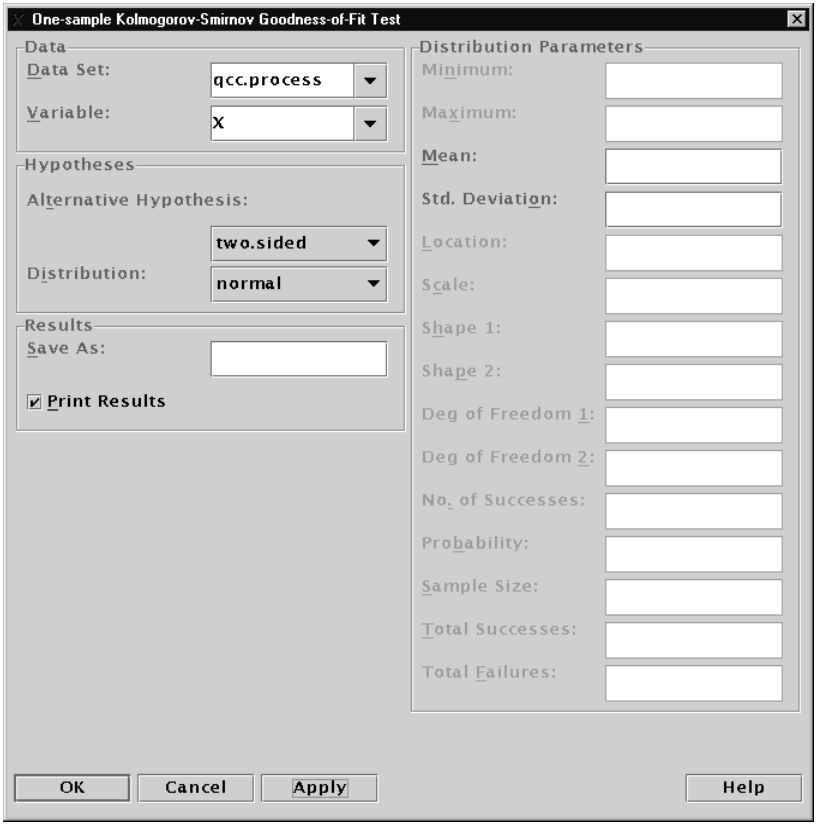


Figure 8.8: *The **One-sample Kolmogorov-Smirnov Goodness-of-Fit Test** dialog.*

Example

We create a data set called `qcc.process` that contains a simulated process with 200 measurements. Ten measurements per day were taken for a total of twenty days. We use the `rnorm` function to generate the data set from a Gaussian distribution.

```
# Use set.seed for reproducibility.
> set.seed(21)
> qcc.process <- data.frame(X = rnorm(200, mean=10),
+ Day = unlist(lapply(1:20,
+ FUN=function(x) rep(x, times=10))))
```

```
> qcc.process

           X Day
 1  9.755851   1
 2  8.959829   1
 3 10.223913   1
 4 10.362865   1
 5  9.477088   1
 6 10.236104   1
 7  8.009497   1
 8 10.213798   1
 9  9.929919   1
10  9.656944   1
11  9.304599   2
12 10.749046   2
13 . . .
```

We can use the Kolmogorov-Smirnov goodness-of-fit test to confirm that `qcc.process` is Gaussian:

1. Open the **One-sample Kolmogorov-Smirnov Goodness-of-Fit Test** dialog. The **Distribution** is **normal** by default.
2. Select `qcc.process` as the **Data Set**.
3. Select `X` as the **Variable**.
4. Click **OK**.

A summary of the goodness-of-fit test appears in the **Report** window. The p-value of 0.5 indicates that we do not reject the hypothesis that the data are normally distributed. The summary also contains estimates of the mean and standard deviation for the distribution.

The **Report** window contains a warning indicating that the Dallal-Wilkinson approximation used in this test is most accurate for extreme p-values (p-values ≤ 0.1). Our actual calculated p-value is 0.776, which is set to 0.5 in the summary to indicate that the null hypothesis is not rejected, but our estimate of the p-value is not highly accurate.

Chi-Square Goodness-of-Fit

The *chi-square goodness-of-fit test* uses Pearson's chi-square statistic to test whether the empirical distribution of a set of observations is consistent with a random sample drawn from a specific theoretical distribution.

Chi-square tests apply to any type of variable: continuous, discrete, or a combination of these. If the hypothesized distribution is discrete and the sample size is large $(n > 50)$, the chi-square is the only valid test. In addition, the chi-square test easily adapts to the situation in which parameters of a distribution are estimated. However, for continuous variables, information is lost by grouping the data.

When the hypothesized distribution is continuous, the Kolmogorov-Smirnov test is more likely than the chi-square test to reject the null hypothesis when it should be rejected. The Kolmogorov-Smirnov test is more powerful than the chi-square test, and hence is preferred for continuous distributions.

Performing Pearson's chi-square test

From the main menu, choose **Statistics ► Compare Samples ► One Sample ► Chi-square GOF**. The **One-sample Chi-Square Goodness-of-Fit Test** dialog opens, as shown in Figure 8.9.

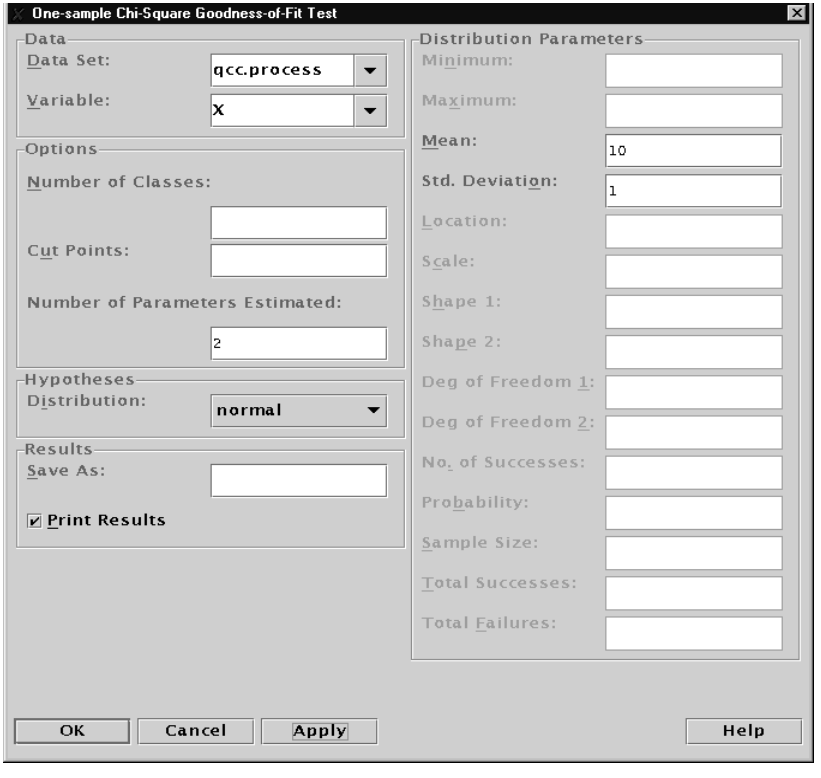


Figure 8.9: *The **One-sample Chi-Square Goodness-of-Fit Test** dialog.*

Example

In the previous section, we created a data set called `qcc.process` that contains a simulated process with 200 measurements. Ten measurements per day were taken for a total of twenty days. We can use the chi-square goodness-of-fit test to confirm that `qcc.process` is Gaussian:

1. If you have not done so already, create the `qcc.process` data set with the instructions given on page 284.
2. Open the **One-sample Chi-Square Goodness-of-Fit Test** dialog. The **Distribution** is **normal** by default.
3. Select `qcc.process` as the **Data Set**.
4. Select `X` as the **Variable**.
5. For the chi-square test, we must specify parameter estimates for the mean and standard deviation of the distribution. Enter 10 as the **Mean** and 1 as the **Std. Deviation**. If you do not know good parameter estimates for your data, you can use the **Summary Statistics** dialog to compute them.
6. Since we are estimating the mean and standard deviation of our data, we should adjust for these parameter estimates when performing the goodness-of-fit test. Enter 2 as the **Number of Parameters Estimated**.
7. Click **OK**.

A summary of the goodness-of-fit test appears in the **Report** window.

Two-Sample Tests

S-PLUS supports a variety of statistical tests for comparing two population parameters. That is, we test the null hypothesis that H_0: $\Theta_1 = \Theta_2$, where Θ_1 and Θ_2 are the two population parameters.

- **Two-sample t-test**: a test to compare two population means μ_1 and μ_2. For small data sets, we require that both populations have a normal distribution. Variations of the two-sample t-test, such as the paired t-test and the two-sample t-test with unequal variances, are also supported.

- **Two-sample Wilcoxon test**: a nonparametric test to compare two population means μ_1 and μ_2. As with the t-test, we test if $\mu_1 = \mu_2$, but we make no distributional assumptions about our populations. Two forms of the Wilcoxon test are supported: the signed rank test and the rank sum test.

- **Kolmogorov-Smirnov goodness-of-fit test**: a test to determine whether two samples come from the same distribution.

Two-Sample t-Test

The *two-sample t-test* is used to test whether two samples come from distributions with the same means. This test handles both paired and independent samples. The samples are assumed to come from Gaussian (normal) distributions. If this is not the case, then a nonparametric test, such as the Wilcoxon rank sum test, may be a more appropriate test of location.

Performing a two-sample t-test

From the main menu, choose **Statistics ▶ Compare Samples ▶ Two Samples ▶ t Test**. The **Two-sample t Test** dialog opens, as shown in Figure 8.10.

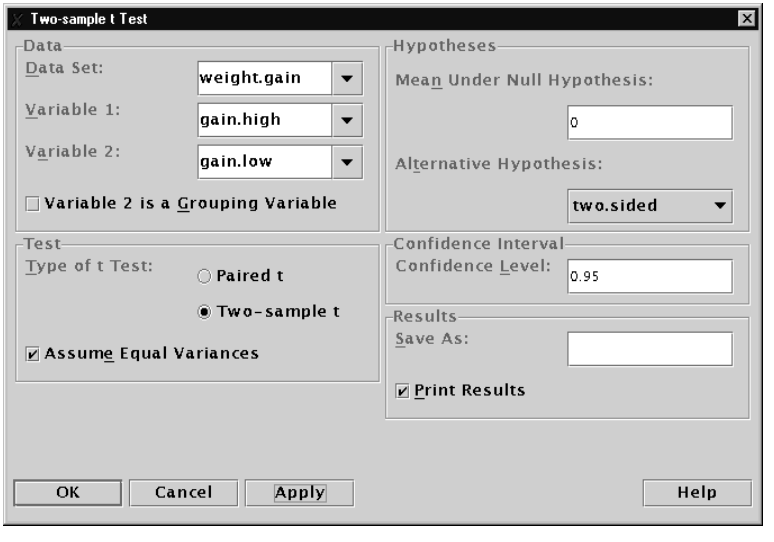


Figure 8.10: *The **Two-sample t Test** dialog.*

Example

Suppose you are a nutritionist interested in the relative merits of two diets, one featuring high protein and the other featuring low protein. Do the two diets lead to differences in mean weight gain? Consider the data in Table 8.1, which shows the weight gains (in grams) for two lots of female rats under the two diets. The first lot, consisting of 12 rats, was given the high-protein diet, and the second lot, consisting of 7 rats, was given the low-protein diet. These data appear in section 6.9 of Snedecor and Cochran (1980).

Table 8.1: *Weight gain data.*

High Protein	Low Protein
134	70
146	118
104	101
119	85
124	107
161	132
107	94
83	
113	
129	
97	
123	

The high-protein and low-protein samples are presumed to have mean-value location parameters μ_H and μ_L, and standard deviation scale parameters σ_H and σ_L, respectively. While you are primarily interested in whether there is any difference in the mean values, you may also be interested in whether the two diets result in different variabilities, as measured by the standard deviations. This example shows you how to use S-PLUS to answer such questions.

Setting up the data

The data consist of two sets of observations, so they are appropriately described in S-PLUS as a data frame with two variables. Since S-PLUS requires data frame columns to be of equal length, we must pad the column representing the low-protein samples with NAs. To create such a data frame, type the following in the **Commands** window:

```
> weight.gain <- data.frame(gain.high=c(134, 146, 104, 119,
+ 124, 161, 107, 83, 113, 129, 97, 123), gain.low=c(70, 118,
+ 101, 85, 107, 132, 94, NA, NA, NA, NA, NA))
> weight.gain

 gain.high gain.low
       134       70
       146      118
       104      101
       119       85
       124      107
       161      132
       107       94
        83       NA
       113       NA
       129       NA
        97       NA
       123       NA
```

Exploratory data analysis

To begin, we want to evaluate the shape of the distribution to see if both our variables are normally distributed. To do this, create the following plots for each of the variables: a boxplot, a histogram, a density plot, and a QQ normal plot. You can create these plots from

the **Graph** menu or from the **Commands** window. We use the function `eda.shape` defined in the section One-Sample t-Test on page 276:

```
> eda.shape(weight.gain$gain.high)
```

The plots that `eda.shape` generates for the high-protein group are shown in Figure 8.11. They indicate that the data come from a nearly normal distribution, and there is no indication of outliers. The plots for the low-protein group, which we do not show, support the same conclusions.

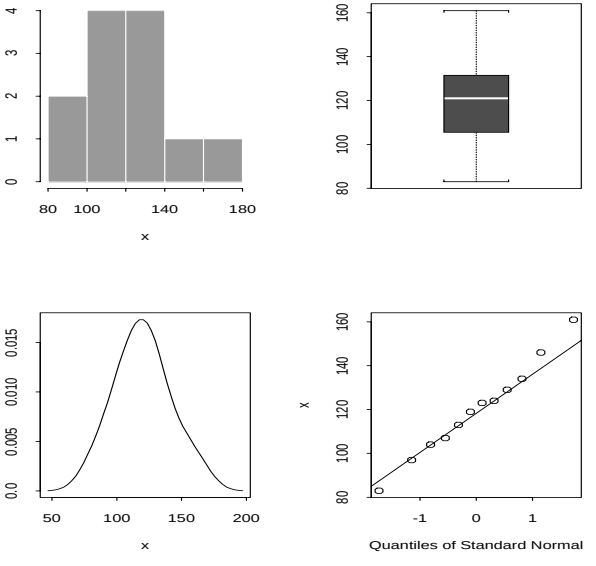


Figure 8.11: *Exploratory data analysis plots for the high-protein diet.*

Statistical inference

Is the mean weight gain the same for the two groups of rats? Specifically, does the high-protein group show a higher average weight gain? From our exploratory data analysis, we have good reason to believe that Student's t-test provides a valid test of our hypotheses. As in the one-sample case, you can obtain confidence intervals and hypothesis test p-values for the difference $\mu_1 - \mu_2$

between the two mean-value location parameters μ_1 and μ_2. To do this, we use the **Two-sample t Test** and **Two-sample Wilcoxon Test** dialogs.

Each two-sample test is specified by a hypothesis to be tested, the confidence level, and a hypothesized μ_0 that refers to the *difference* of the two sample means. However, because of the possibility that the two samples may be from different distributions, you may also specify whether the two samples have equal variances. To determine the correct setting for the option **Assume Equal Variances**, you can either use informal inspection of the variances and box plots, or conduct a formal *F*-test to check for equality of variance. If the heights of the boxes in the two box plots are approximately the same, then so are the variances of the two samples. In the `weight.gain` example, the box plots indicate that the equal variance assumption probably holds. To check this assumption, we calculate the variances exactly:

1. Open the **Summary Statistics** dialog.
2. Enter `weight.gain` as the **Data Set**.
3. Click on the **Statistics** tab, and select the **Variance** check box.
4. Click **OK**.

The following output appears in the **Report** window:

```
      ***  Summary Statistics for data in:  weight.gain ***

           gain.high  gain.low
     Min:  83.00000  70.00000
 1st Qu.: 106.25000  89.50000
    Mean: 120.00000 101.00000
  Median: 121.00000 101.00000
 3rd Qu.: 130.25000 112.50000
     Max: 161.00000 132.00000
 Total N:  12.00000  12.00000
   NA's :   0.00000   5.00000
Variance: 457.45455 425.33333
 Std Dev.:  21.38819  20.62361
```

The actual variances of our two samples are 457.4 and 425.3, respectively. These values support our assertion of equal variances.

We are interested in two alternative hypotheses: the two-sided alternative that $\mu_H - \mu_L = 0$ and the one-sided alternative that $\mu_H - \mu_L > 0$. To test these, we run the standard two-sample t-test twice, once with the default two-sided alternative and a second time with the one-sided alternative hypothesis `greater`.

1. Open the **Two-sample t Test** dialog.
2. Type `weight.gain` in the **Data Set** field.
3. Select `gain.high` as **Variable 1** and `gain.low` as **Variable 2**. By default, the **Variable 2 is a Grouping Variable** check box should not be selected, and the **Assume Equal Variances** check box should be selected.
4. Click **Apply**.

The result appears in the **Report** window:

```
        Standard Two-Sample t-Test

data:  x: gain.high in weight.gain , and y: gain.low
    in weight.gain
t = 1.8914, df = 17, p-value = 0.0757
alternative hypothesis: true difference in means is
    not equal to 0
95 percent confidence interval:
  -2.193679  40.193679
sample estimates:
 mean of x mean of y
       120       101
```

The p-value is 0.0757, so the null hypothesis is rejected at the 0.10 level but not at the 0.05 level. The confidence interval is $(-2.2, 40.2)$. In other words, we conclude at the 0.05 level that there is no significant difference in the weight gain between the two diets.

To test the one-sided alternative that $\mu_H - \mu_L > 0$, we change the **Alternative Hypothesis** field to `greater` in the **Two-sample t Test** dialog. Click **OK** to perform the test and see the output shown below.

```
        Standard Two-Sample t-Test

data:  x: gain.high in weight.gain , and y: gain.low
   in weight.gain
t = 1.8914, df = 17, p-value = 0.0379
alternative hypothesis: true difference in means is
   greater than 0
95 percent confidence interval:
 1.525171       NA
sample estimates:
 mean of x mean of y
       120       101
```

In this case, the p-value is just half of the p-value for the two-sided alternative. This relationship between the p-values holds in general. You also see that when you use the `greater` alternative hypothesis, you get a lower confidence bound. This is the natural one-sided confidence interval corresponding to the "greater than" alternative.

Two-Sample Wilcoxon Test

The *Wilcoxon rank sum test* is used to test whether two sets of observations come from the same distribution. The alternative hypothesis is that the observations come from distributions with identical shape but different locations. Unlike the two-sample t-test, this test does not assume that the observations come from normal (Gaussian) distributions. The Wilcoxon rank sum test is equivalent to the Mann-Whitney test.

For paired data, specify "signed rank" as the type of Wilcoxon rank test.

Performing a two-sample Wilcoxon rank test

From the main menu, choose **Statistics ▶ Compare Samples ▶ Two Samples ▶ Wilcoxon Rank Test**. The **Two-sample Wilcoxon Test** dialog opens, as shown in Figure 8.12.

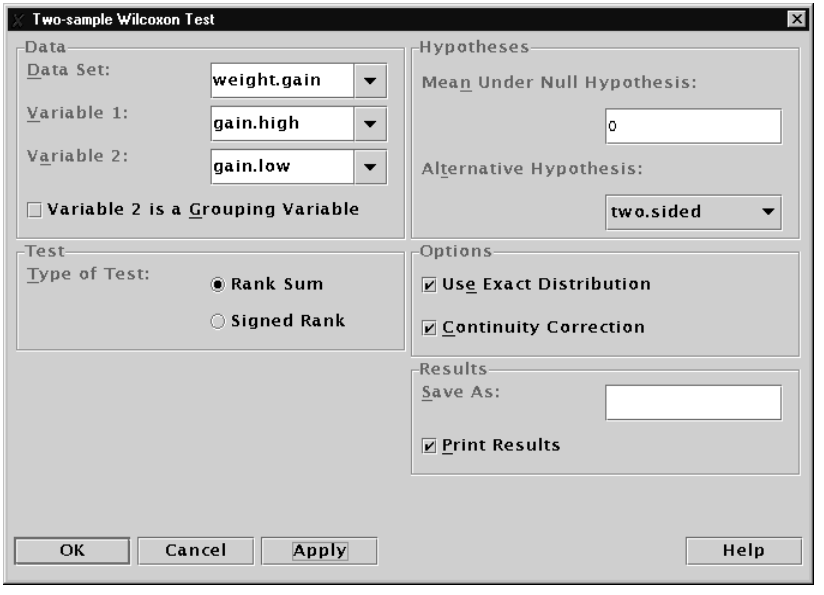


Figure 8.12: *The **Two-sample Wilcoxon Test** dialog.*

Example

In the section Two-Sample t-Test on page 288, we conducted a test to see if the mean weight gain from a high-protein diet differs from that of a low-protein diet. The two-sample t-test was significant at the 0.10 level but not at the 0.05 level. Since normality holds, a two-sample t-test is probably most appropriate for these data. However, for illustrative purposes we conduct a two-sample Wilcoxon test to see if the two diets differ in mean weight gain. We conduct a two-sided test, where the null hypothesis is that the difference in diets is 0; that is, we test if the mean weight gain is the same for each diet.

1. If you have not done so already, create the `weight.gain` data set with the instructions given on page 290.
2. Open the **Two-sample Wilcoxon Test** dialog.

3. Specify `weight.gain` as the **Data Set**.
4. Select `gain.high` as **Variable 1** and `gain.low` as **Variable 2**. By default, the **Variable 2 is a Grouping Variable** check box should not be selected, and the **Type of Rank Test** should be set to **Rank Sum**. Click **OK**.

The **Report** window shows the following output:

```
        Wilcoxon rank-sum test
data:  x: gain.high in weight.gain , and y: gain.low in
  weight.gain
rank-sum normal statistic with correction Z = 1.6911,
  p-value = 0.0908
alternative hypothesis: true mu is not equal to 0
```

You may also see a warning in the **Report** window because the value 107 appears twice in the data set. The warning can be ignored for now. The p-value of 0.0908 is based on the normal approximation, which is used because of ties in the data. It is close to the t-statistic p-value of 0.0757. It therefore supports our conclusion that the mean weight gain is not significantly different at level 0.05 in the high- and low-protein diets.

Kolmogorov-Smirnov Goodness-of-Fit

The *two-sample Kolmogorov-Smirnov goodness-of-fit test* is used to test whether two sets of observations could reasonably have come from the same distribution. This test assumes that the two samples are random and mutually independent, and that the data are measured on at least an ordinal scale. In addition, the test gives exact results only if the underlying distributions are continuous.

Perform a two-sample Kolmogorov-Smirnov goodness-of-fit test

From the main menu, choose **Statistics ▶ Compare Samples ▶ Two Samples ▶ Kolmogorov-Smirnov GOF**. The **Two-sample Kolmogorov-Smirnov Goodness-of-Fit Test** dialog opens, as shown in Figure 8.13.

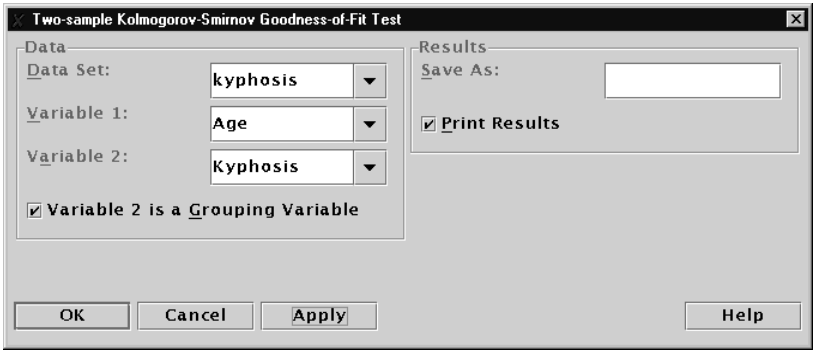


Figure 8.13: *The **Two-sample Kolmogorov-Smirnov Goodness-of-Fit Test** dialog.*

Example

The `kyphosis` data set has 81 rows representing data on 81 children who have had corrective spinal surgery. The outcome `Kyphosis` is a binary variable, and the other three columns `Age`, `Number`, and `Start`, are numeric. Kyphosis is a post-operative deformity which is present in some children receiving spinal surgery. We are interested in examining whether the child's age, the number of vertebrae operated on, or the starting vertebra influence the likelihood of the child having a deformity. As an exploratory tool, we test whether the distributions of `Age`, `Number`, and `Start` are the same for the children with and without kyphosis.

1. Open the **Two-sample Kolmogorov-Smirnov Goodness-of-Fit Test** dialog.
2. Type `kyphosis` in the **Data Set** field.
3. We perform separate tests for each of the three covariates, in each case grouping by `Kyphosis`. Select `Kyphosis` as **Variable 2**. Select the **Variable 2 is a Grouping Variable** check box.
4. Select `Age` as **Variable 1**. Click **Apply**.
5. Select `Number` as **Variable 1**. Click **Apply**.
6. Select `Start` as **Variable 1**. Click **OK**.

A **Report** window appears with three goodness-of-fit summaries. The p-values for `Age`, `Number`, and `Start` are 0.076, 0.028, and 0.0002, respectively. This suggests that the children with and without kyphosis do not differ significantly in the distribution of their ages, but do differ significantly in the distributions of how many vertebrae were involved in the operation, as well as which vertebra was the starting vertebra. This is consistent with the logistic regression model fit to these data later, in the section Logistic Regression on page 356.

K-Sample Tests

S-PLUS supports a variety of techniques to analyze group mean differences in designed experiments.

- **One-way analysis of variance**: a simple one-factor analysis of variance. No interactions are assumed among the main effects. That is, the k samples are considered independent, and the data must be normally distributed.
- **Kruskal-Wallis rank sum test**: a nonparametric alternative to a one-way analysis of variance. No distributional assumptions are made.
- **Friedman rank sum test**: a nonparametric analysis of means of a one-factor designed experiment with an unreplicated blocking variable.

The **ANOVA** dialog provides analysis of variance models involving more than one factor; see the section Analysis of Variance on page 361.

One-Way Analysis of Variance

The **One-Way Analysis of Variance** dialog generates a simple analysis of variance (ANOVA) table when there is a grouping variable available that defines separate samples of the data. No interactions are assumed among the main effects; that is, the samples are considered to be independent. The ANOVA tables include F-statistics, which test whether the mean values for all of the groups are equal. These statistics assume that the observations are normally (Gaussian) distributed.

For more complex models or ANOVA with multiple predictors, use the **Analysis of Variance** dialog.

Perform a one-way ANOVA

From the main menu, choose **Statistics ▶ Compare Samples ▶ k Samples ▶ One-way ANOVA**. The **One-way Analysis of Variance** dialog opens, as shown in Figure 8.14.

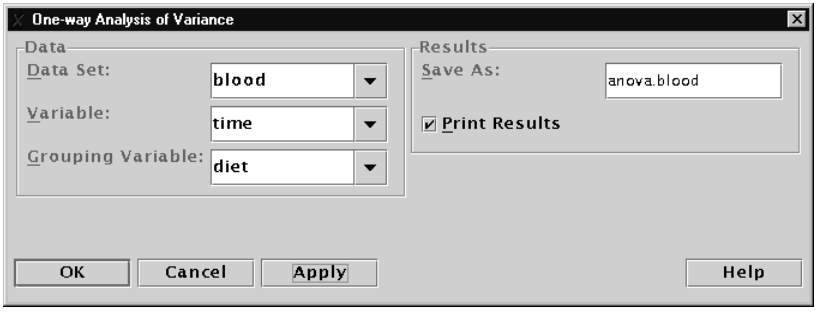


Figure 8.14: *The **One-way Analysis of Variance** dialog.*

Example

The simplest kind of experiments are those in which a single continuous *response* variable is measured a number of times for each of several *levels* of some experimental *factor*. For example, consider the data in Table 8.2 (from Box, Hunter, and Hunter (1978)). The data consist of numerical values of blood coagulation times for each of four diets. Coagulation time is the continuous response variable, and diet is a *qualitative* variable, or *factor*, having four levels: A, B, C, and D. The diets corresponding to the levels A, B, C, and D were determined by the experimenter.

Your main interest is to see whether or not the factor "diet" has any effect on the mean value of blood coagulation time. Experimental factors such as "diet" are often called the *treatments*.

Formal statistical testing for whether the factor levels affect the mean coagulation time is carried out using analysis of variance (ANOVA). This method needs to be complemented by exploratory graphics to provide confirmation that the model assumptions are sufficiently correct to validate the formal ANOVA conclusion. S-PLUS provides tools for you to do both the data exploration and the formal ANOVA.

Table 8.2: *Blood coagulation times for four diets.*

Diet			
A	**B**	**C**	**D**
62	63	68	56
60	67	66	62
63	71	71	60
59	64	67	61
	65	68	63
	66	68	64
			63
			59

Setting up the data

We have one factor variable `diet` and one response variable `time`. The data are appropriately described in S-PLUS as a data set with two columns. The data presented in Table 8.2 can be generated by typing the following in the **Commands** window:

```
> diet <- factor(c(rep("A",4), rep("B",6), rep("C",6),
+ rep("D",8)))

> Time <- scan()
1: 62 60 63 59
5: 63 67 71 64 65 66
11: 68 66 71 67 68 68
17: 56 62 60 61 63 64 63 59
25:
```

```
> blood <- data.frame(diet=diet, time=Time)
> blood

   diet time
 1    A   62
 2    A   60
 3    A   63
 4    A   59
 5    B   63
 6    B   67
 7    B   71
 8    B   64
 9    B   65
10    B   66
11    C   68
12    C   66
13    C   71
14    C   67
15    C   68
16    C   68
17    D   56
18    D   62
19    D   60
20    D   61
21    D   63
22    D   64
23    D   63
24    D   59
```

Exploratory data analysis

Box plots are a quick and easy way to get a first look at the data:

```
> boxplot(split(blood$time, blood$diet),
+ xlab="diet", ylab="time")
```

The resulting box plots are similar to those in Figure 8.15. This plot indicates that the responses for diets A and D are quite similar, while the median responses for diets B and C are considerably larger relative to the variability reflected by the heights of the boxes. Thus, you suspect that diet has an effect on blood coagulation time.

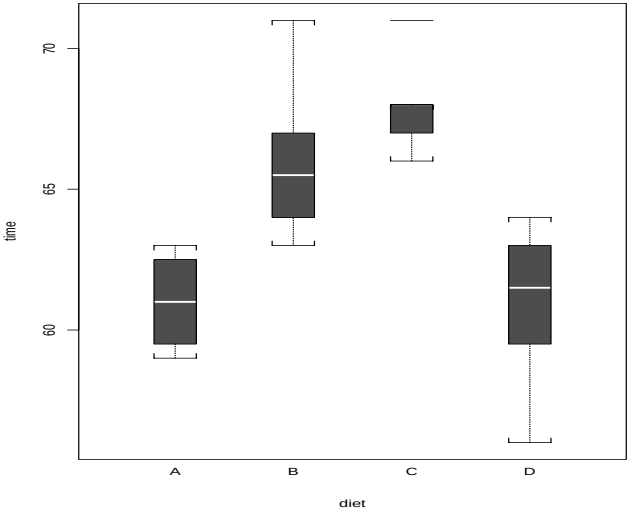


Figure 8.15: *Box plots for each of the four diets in the* `blood` *data set.*

The one-way layout model and analysis of variance

The classical model for experiments with a single factor is

$$y_{ij} = \mu_i + \varepsilon_{ij} \qquad \begin{aligned} j &= 1, \ldots, J_i \\ i &= 1, \ldots, I \end{aligned}$$

where μ_i is the mean value of the response for the ith level of the experimental factor. There are I levels of the experimental factor, and J_i measurements $y_{i1}, y_{i2}, \ldots, y_{iJ_i}$ are taken on the response variable for level i of the experimental factor. Using the treatment terminology, there are I treatments and μ_i is called the ith treatment mean. The is often called the *one-way layout* model. For the blood coagulation experiment, there are $I = 4$ diets, and the means μ_1,

μ_2, μ_3, and μ_4 correspond to diets A, B, C, and D, respectively. The numbers of observations are $J_A = 4$, $J_B = 6$, $J_C = 6$, and $J_D = 8$.

You may carry out the analysis of variance using the **One-way Analysis of Variance** dialog.

1. Open the **One-way Analysis of Variance** dialog.
2. Type `blood` in the **Data Set** field.
3. Select `time` as the **Variable** and `diet` as the **Grouping Variable**.
4. To generate multiple comparisons in a later section, we save the results by typing `anova.blood` in the **Save As** field.
5. Click **OK** to perform the ANOVA.

The results are displayed in the **Report** window:

```
   ***  One-Way ANOVA for data in time by diet ***
Call:
   aov(formula = time ~ diet, data = blood)
Terms:
                 diet Residuals
 Sum of Squares  228       112
Deg. of Freedom    3        20
Residual standard error: 2.366432
Estimated effects may be unbalanced
          Df Sum of Sq Mean Sq  F Value         Pr(F)
     diet  3       228    76.0 13.57143 0.00004658471
Residuals 20       112     5.6
```

The p-value is equal to 0.000047, which is highly significant; we therefore conclude that diet does affect blood coagulation times.

Kruskal-Wallis Rank Sum Test

The *Kruskal-Wallis rank test* is a nonparametric alternative to a one-way analysis of variance. The null hypothesis is that the true location parameter for y is the same in each of the groups. The alternative hypothesis is that y is different in at least one of the groups. Unlike one-way ANOVA, this test does not require normality.

Performing a Kruskal-Wallis rank sum test

From the main menu, choose **Statistics ▶ Compare Samples ▶ k Samples ▶ Kruskal-Wallis Rank Test**. The **Kruskal-Wallis Rank Sum Test** dialog opens, as shown in Figure 8.16.

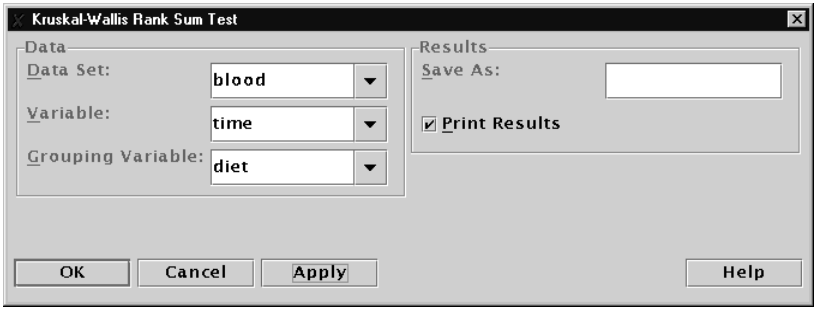


Figure 8.16: *The **Kruskal-Wallis Rank Sum Test** dialog.*

Example

In the section One-Way Analysis of Variance on page 298, we concluded that diet affects blood coagulation times. The one-way ANOVA requires the data to be normally distributed. The nonparametric Kruskal-Wallis rank sum test does not make any distributional assumptions and can be applied to a wider variety of data. We now conduct the Kruskal-Wallis rank sum test on the `blood` data set.

1. If you have not done so already, create the `blood` data set with the instructions given on page 300.
2. Open the **Kruskal-Wallis Rank Sum Test** dialog.
3. Type `blood` in the **Data Set** field.
4. Select `time` as the **Variable** and `diet` as the **Grouping Variable**, and click **OK**.

The **Report** window displays the result:

```
        Kruskal-Wallis rank sum test
data:  time and diet from data set blood
Kruskal-Wallis chi-square = 17.0154, df = 3,
  p-value = 0.0007
alternative hypothesis: two.sided
```

The p-value is 0.0007, which is highly significant. The Kruskal-Wallis rank sum test confirms the results of our one-way ANOVA.

Friedman Rank Test

The *Friedman rank test* is appropriate for data arising from an unreplicated complete block design. In these kinds of designs, exactly one observation is collected from each experimental unit, or *block*, under each treatment. The elements of y are assumed to consist of a groups effect, plus a blocks effect, plus independent and identically distributed residual errors. The interaction between groups and blocks is assumed to be zero.

In the context of a two-way layout with factors groups and blocks, a typical null hypothesis is that the true location parameter for y, net of the blocks effect, is the same in each of the groups. The alternative hypothesis is that it is different in at least one of the groups.

Performing a Friedman rank test

From the main menu, choose **Statistics ▶ Compare Samples ▶ k Samples ▶ Friedman Rank Test**. The **Friedman Rank Sum Test** dialog opens, as shown in Figure 8.17.

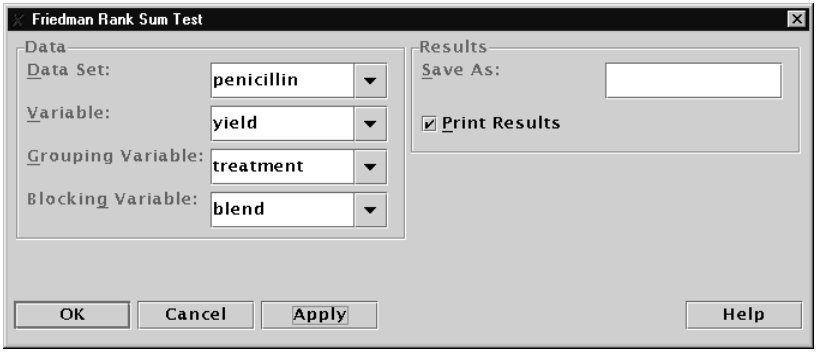


Figure 8.17: *The **Friedman Rank Sum Test** dialog.*

Example

The data set shown in Table 8.3 was first used by Box, Hunter, and Hunter in 1978. The data was collected to determine the effect of treatments A, B, C, and D on the yield of penicillin in a penicillin manufacturing process. The response variable is `yield`, and the

treatment variable is `treatment`. There is a second factor, `blend`, since a separate blend of the corn-steep liquor had to be made for each application of the treatments.

Our main interest is in determining whether the `treatment` factor affects `yield`. The `blend` factor is of only secondary interest; it is a blocking variable introduced to increase the sensitivity of the inference for treatments. The order of the treatments within blocks was chosen at random. Hence, this is a randomized block experiment.

Table 8.3: *The effect of four treatments on the yield of penicillin.*

blend	treatment	yield
1	A	89
2	A	84
3	A	81
4	A	87
5	A	79
1	B	88
2	B	77
3	B	87
4	B	92
5	B	81
1	C	97
2	C	92
3	C	87
4	C	89
5	C	80
1	D	94
2	D	79
3	D	85
4	D	84
5	D	88

Setting up the data

To create a `penicillin` data set containing the information in Table 8.3, type the following in the **Commands** window:

```
> blend <- factor(rep(c("Blend 1", "Blend 2", "Blend 3",
+ "Blend 4", "Blend 5"), times=4))

> treatment <- factor(c(rep("A",5), rep("B",5), rep("C",5),
rep("D",5)))

> yield <- scan()
1: 89 84 81 87 79
6: 88 77 87 92 81
11: 97 92 87 89 80
16: 94 79 85 84 88
21:

> penicillin <- data.frame(blend, treatment, yield)
> penicillin

     blend treatment yield
 1 Blend 1         A    89
 2 Blend 2         A    84
 3 Blend 3         A    81
 4 Blend 4         A    87
 5 Blend 5         A    79
 6 Blend 1         B    88
 7 Blend 2         B    77
 8 . . .
```

Statistical inference

We use the Friedman rank test to test the null hypothesis that there is no treatment effect.

1. Open the **Friedman Rank Sum Test** dialog.
2. Type `penicillin` in the **Data Set** field.
3. Select `yield` as the **Variable**, `treatment` as the **Grouping Variable**, and `blend` as the **Blocking Variable**.
4. Click **OK**.

A summary for the Friedman test appears in the **Report** window. The p-value is 0.322, which is not significant. This p-value is computed using an asymptotic chi-squared approximation.

Counts and Proportions

S-PLUS supports a variety of techniques to analyze counts and proportions.

- **Binomial Test:** an exact test used with binomial data to assess whether the data come from a distribution with a specified proportion parameter.
- **Proportions Parameters:** a chi-square test to assess whether a binomial sample has a specified proportion parameter, or whether two binomial samples have the same proportion parameter.
- **Fisher's Exact Test:** a test for independence between the rows and columns of a contingency table.
- **McNemar's Test:** a test for independence in a contingency table when matched variables are present.
- **Mantel-Haenszel Test:** a chi-square test of independence for a three-dimensional contingency table.
- **Chi-square Test:** a chi-square test for independence for a two-dimensional contingency table.

Binomial data are data representing a certain number k of successes out of n trials, where observations occur independently with probability p of a success. Contingency tables contain counts of the number of occurrences of each combination of two or more categorical (factor) variables.

Binomial Test

The *exact binomial test* is used with binomial data to assess whether the data are likely to have come from a distribution with a specified proportion parameter p. Binomial data are data representing a certain number k of successes out of n trials, where observations occur independently with probability p of a success. Examples include coin toss data.

Performing an exact binomial test

From the main menu, choose **Statistics ▶ Compare Samples ▶ Counts and Proportions ▶ Binomial Test**. The **Exact Binomial Test** dialog opens, as shown in Figure 8.18.

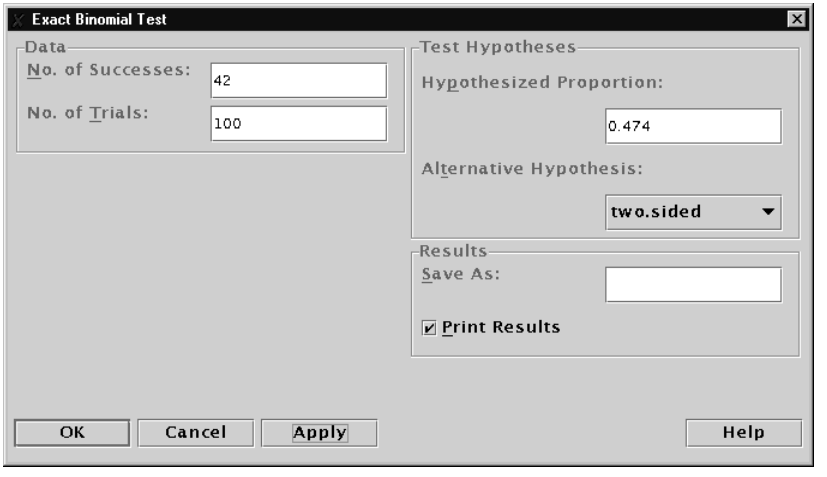


Figure 8.18: *The **Exact Binomial Test** dialog.*

Example

When you play roulette and bet on red, you expect your probability of winning to be close to, but slightly less than 0.5. You expect this because, in the United States, a roulette wheel has 18 red slots, 18 black slots, and two additional slots labeled "0" and "00." This gives a total of 38 slots into which the ball can fall. Thus, for a fair or perfectly balanced wheel, you expect the probability of red to be $p_0 = 18/38 = 0.474$. You hope that the house is not cheating you by altering the roulette wheel so that the probability of red is less than 0.474.

For example, suppose you bet on red 100 times and red comes up 42 times. You wish to ascertain whether these results are reasonable with a fair roulette wheel.

1. Open the **Exact Binomial Test** dialog.
2. Enter 42 as the **No. of Successes**. Enter 100 as the N**o. of Trials**.

3. Enter 0.474 as the **Hypothesized Proportion**.
4. Click **OK**.

A summary of the test appears in the **Report** window. The p-value of 0.3168 indicates that our sample is consistent with data drawn from a binomial distribution with a proportions parameter of 0.474. Hence, the roulette wheel seems to be fair.

Proportions Parameters

The *proportions parameters test* uses a Pearson's chi-square statistic to assess whether a binomial sample has a specified proportion parameter p. In addition, it can assess whether two or more samples have the same proportion parameter. As the proportions parameters test uses a normal approximation to the binomial distribution, it is less powerful than the exact binomial test. Hence, the exact binomial test is usually preferred. The advantages of the proportions parameters test are that it provides a confidence interval for the proportions parameter, and that it may be used with multiple samples.

Performing a proportions parameters test

From the main menu, choose **Statistics ▶ Compare Samples ▶ Counts and Proportions ▶ Proportions Parameters**. The **Proportions Test** dialog opens, as shown in Figure 8.19.

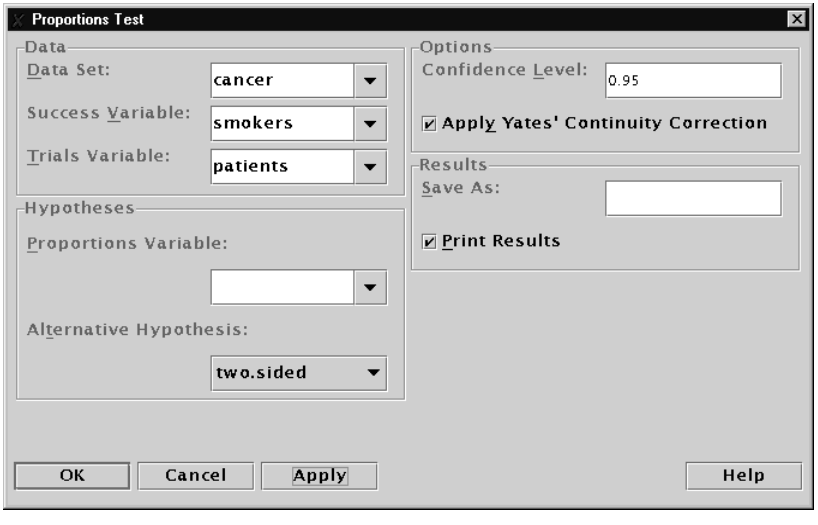


Figure 8.19: *The **Proportions Test** dialog.*

Example

Sometimes you may have multiple samples of subjects, with each subject characterized by the presence or absence of some characteristic. An alternative, but equivalent, terminology is that you have three or more sets of trials, with each trial resulting in a success or failure. For example, the data set shown in Table 8.4 summarizes the results of four different studies of lung cancer patients, as presented by Fleiss (1981). Each study has a certain number of `patients`, and for each study a certain number of the patients were `smokers`.

Table 8.4: *Four different studies of lung cancer patients.*

smokers	patients
83	86
90	93
129	136
70	82

Setting up the data

To create a `cancer` data set containing the information in Table 8.4, type the following in the **Commands** window:

```
> cancer <- data.frame(smokers = c(83, 90, 129, 70),
+ patients = c(86, 93, 136, 82))
> cancer

  smokers patients
1      83       86
2      90       93
3     129      136
4      70       82
```

Statistical inference

For the `cancer` data, we are interested in whether the probability of a patient being a smoker is the same in each of the four studies. That is, we wish to test whether each of the studies involve patients from a homogeneous population.

1. Open the **Proportions Test** dialog.
2. Type `cancer` in the **Data Set** field.
3. Select `smokers` as the **Success Variable** and `patients` as the **Trial Variable**.
4. Click **OK**.

A summary of the test appears in the **Report** window. The p-value of 0.0056 indicates that we reject the null hypothesis of equal proportions parameters. Hence, we cannot conclude that all groups have the same probability that a patient is a smoker.

Fisher's Exact Test

Fisher's exact test is a test for independence between the row and column variables of a contingency table. When the data consist of two categorical variables, a contingency table can be constructed reflecting the number of occurrences of each factor combination. Fisher's exact test assesses whether the value of one factor is independent of the value of the other. For example, this might be used to test whether political party affiliation is independent of gender. Certain types of homogeneity, for example, homogeneity of proportions in a $k \times 2$ table, are equivalent to the independence hypothesis. Hence, this test may also be of interest in such cases.

As this is an exact test, the total number of counts in the cross-classification table cannot be greater than 200. In such cases, the chi-square test of independence is preferable.

Performing Fisher's exact test

From the main menu, choose **Statistics ▶ Compare Samples ▶ Counts and Proportions ▶ Fisher's Exact Test**. The **Fisher's Exact Test** dialog opens, as shown in Figure 8.20.

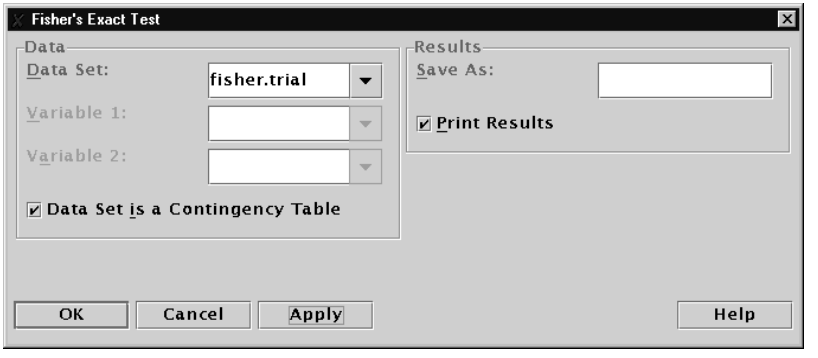


Figure 8.20: *The **Fisher's Exact Test** dialog.*

Example

The data set shown in Table 8.5 contains a contingency table summarizing the results of a clinical trial. Patients were divided into a treatment group which received an experimental drug, and a control group which did not. These patients were then monitored for 28 days, with their survival status noted at the end of the study.

Table 8.5: *A contingency table summarizing the results of a clinical trial.*

	Control	Treated
Died	17	7
Survived	29	38

Setting up the data

To create a `fisher.trial` data set containing the information in Table 8.5, type the following in the **Commands** window:

```
> fisher.trial <- data.frame(c(17,29), c(7,38),
+ row.names=c("Died", "Survived"))
> names(fisher.trial) <- c("Control", "Treated")
```

```
> fisher.trial

          Control Treated
    Died       17       7
Survived       29      38
```

Statistical inference

We are interested in examining whether the treatment affected the probability of survival.

1. Open the **Fisher's Exact Test** dialog.
2. Type `fisher.trial` in the **Data Set** field.
3. Select the **Data Set is a Contingency Table** check box.
4. Click **OK**.

A summary of the test appears in the **Report** window. The p-value of 0.0314 indicates that we reject the null hypothesis of independence. Hence, we conclude that the treatment affects the probability of survival.

McNemar's Test

In some experiments with two categorical variables, one of the variables specifies two or more groups of individuals that receive different treatments. In such situations, matching of individuals is often carried out in order to increase the precision of statistical inference. However, when matching is carried out, the observations usually are not independent. In such cases, the inference obtained from the chi-square test, Fisher's exact test, and Mantel-Haenszel test is not valid because these tests all assume independent observations.

McNemar's test allows you to obtain a valid inference for experiments where matching is carried out. McNemar's statistic is used to test the null hypothesis of symmetry: namely, that the probability of an observation being classified into cell $[i,j]$ is the same as the probability of being classified into cell $[j,i]$. The returned p-value should be interpreted carefully. Its validity depends on the assumption that the cell counts are at least moderately large. Even when cell counts are adequate, the chi-square is only a large-sample approximation to the true distribution of McNemar's statistic under the null hypothesis.

Performing McNemar's test

From the main menu, choose **Statistics ▶ Compare Samples ▶ Counts and Proportions ▶ McNemar's Test**. The **McNemar's Chi-Square Test** dialog opens, as shown in Figure 8.21.

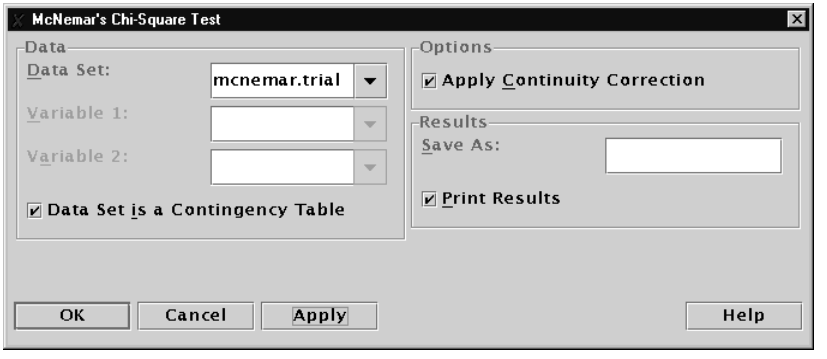


Figure 8.21: *The **McNemar's Chi-Square Test** dialog.*

Example

The data set shown in Table 8.6 contains a contingency table of matched pair data, in which each count is associated with a matched pair of individuals.

Table 8.6: *Contingency table of matched pair data.*

	B,Survive	B,Die
A,Survive	90	16
A,Die	5	510

In this table, each entry represents a pair of patients, one of whom was given treatment A while the other was given treatment B. For instance, the 5 in the lower left cell means that in five pairs, the person with treatment A died, while the individual the person was paired with survived. We are interested in the relative effectiveness of treatments A and B in treating a rare form of cancer.

A pair in the table for which one member of a matched pair survives while the other member dies is called a *discordant pair*. There are 16 discordant pairs in which the individual who received treatment A survived and the individual who received treatment B died. There are five discordant pairs with the reverse situation, in which the individual who received treatment A died and the individual who received treatment B survived. If both treatments are equally effective, then we expect these two types of discordant pairs to occur with nearly equal frequency. Put in terms of probabilities, the null hypothesis is that $p_1 = p_2$, where p_1 is the probability that the first type of discordancy occurs and p_2 is the probability that the second type of discordancy occurs.

Setting up the data

To create a `mcnemar.trial` data set containing the information in Table 8.6, type the following in the **Commands** window:

```
> mcnemar.trial <- data.frame(c(90,5), c(16,510),
+ row.names=c("A.Survive", "A.Die"))
> names(mcnemar.trial) <- c("B.Survive", "B.Die")
> mcnemar.trial

          B.Survive B.Die
A.Survive        90    16
    A.Die         5   510
```

Statistical inference

We use McNemar's test to examine whether the treatments are equally effective.

1. Open the **McNemar's Square Test** dialog.
2. Type `mcnemar.trial` in the **Data Set** field.
3. Select the **Data Set is a Contingency Table** check box.
4. Click **OK**.

A summary of the test appears in the **Report** window. The p-value of 0.0291 indicates that we reject the null hypothesis of symmetry in the table. This suggests that the two treatments differ in their efficacy.

Mantel-Haenszel Test

The *Mantel-Haenszel test* performs a chi-square test of independence on a three-dimensional contingency table. It is used for a contingency table constructed from three factors. As with McNemar's test, the returned p-value should be interpreted carefully. Its validity depends on the assumption that certain sums of expected cell counts are at least moderately large. Even when cell counts are adequate, the chi-square is only a large-sample approximation to the true distribution of the Mantel-Haenszel statistic under the null hypothesis.

Performing a Mantel-Haenszel test

From the main menu, choose **Statistics ▶ Compare Samples ▶ Counts and Proportions ▶ Mantel-Haenszel Test**. The **Mantel-Haenszel's Chi-Square Test** dialog opens, as shown in Figure 8.22.

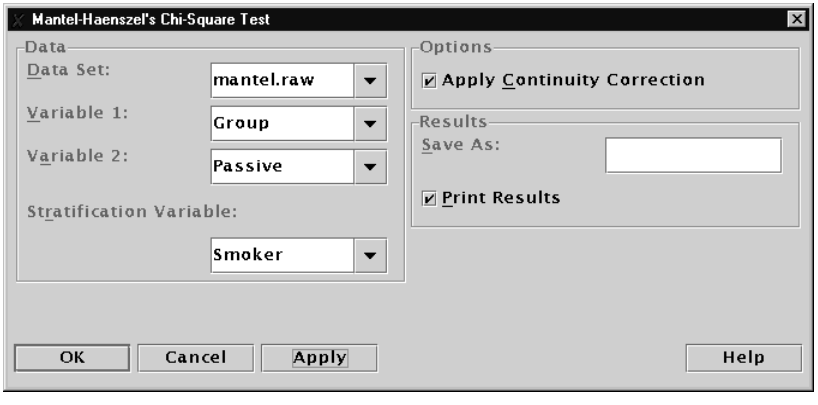


Figure 8.22: *The **Mantel-Haenszel's Chi-Square Test** dialog.*

Example

The data set shown in Table 8.7 contains a three-way contingency table summarizing the results from a cancer study. The first column indicates whether an individual is a smoker. In the second column, "Case" refers to an individual who had cancer and "Control" refers to an individual who did not have cancer. The third column indicates whether an individual is a *passive smoker*. A passive smoker is a person who lives with a smoker, so it is therefore possible for a person to be considered both a smoker and a passive smoker. The fourth column indicates the number of individuals with each combination of `Smoker`, `Group`, and `Passive` values.

Table 8.7: *A three-way contingency table summarizing the results of a cancer study.*

Smoker	Group	Passive	Number
Yes	Case	Yes	120
Yes	Case	No	111
Yes	Control	Yes	80
Yes	Control	No	155
No	Case	Yes	161
No	Case	No	117
No	Control	Yes	130
No	Control	No	124

We are primarily interested in whether passive smoke influences the likelihood of getting cancer. However, smoking status could be a confounding variable because both smoking and passive smoking are related to the outcome, cancer status. We would like to use the information on smoking status to produce an overall test of independence between cancer status and passive smoking status. You can do so for two or more 2×2 tables with the Mantel-Haenszel test.

Setting up the data

To create a `mantel.trial` data set containing the information in Table 8.7, type the following in the **Commands** window:

```
> mantel.trial <- data.frame(
+ Smoker = factor(c(rep("Yes",4), rep("No",4))),
+ Group = factor(c("Case", "Case", "Control", "Control",
+ "Case", "Case", "Control", "Control")),
+ Passive = factor(c("Yes", "No", "Yes", "No", "Yes", "No",
+ "Yes", "No")),
+ Number = c(120, 111, 80, 155, 161, 117, 130, 124))
```

```
> mantel.trial

  Smoker   Group Passive Number
1    Yes    Case     Yes    120
2    Yes    Case      No    111
3    Yes Control     Yes     80
4    Yes Control      No    155
5     No    Case     Yes    161
6     No    Case      No    117
7     No Control     Yes    130
8     No Control      No    124
```

The `mantel.trial` data set has eight rows representing the eight possible combinations of three factors with two levels each. However, the **Mantel-Haenszel Chi-Square Test** dialog requires data to be in its raw form, and does not accept data in a contingency table. To recreate the raw data, type the following in the **Commands** window:

```
> mantel.raw <- mantel.trial[rep(1:8,mantel.trial$Number),]
```

This replicates each of the integers 1 to 8 as many times as indicated by the corresponding count in the `Number` column. We use the `mantel.raw` data frame in the example analysis below.

Statistical inference

We use the **Mantel-Haenszel Chi-Square Test** dialog to test the independence between cancer status and passive smoking status.

1. Open the **Mantel-Haenszel's Chi-Square Test** dialog.
2. Type `mantel.raw` in the **Data Set** field.
3. Select `Group` as **Variable 1**, `Passive` as **Variable 2**, and `Smoker` as the **Stratification Variable**.
4. Click **OK**.

A summary of the test appears in the **Report** window. The p-value of 0.0002 indicates that we reject the null hypothesis of independence between cancer status and passive smoking.

Chi-Square Test

The chi-square test performs a Pearson's chi-square test on a two-dimensional contingency table. This test is relevant to several types of null hypotheses: statistical independence of the rows and columns, homogeneity of groups, etc. The appropriateness of the test to a

particular null hypothesis and the interpretation of the results depend on the nature of the data at hand. In particular, the sampling scheme is important in determining the appropriate of a chi-square test.

The p-value returned by a chi-square test should be interpreted carefully. Its validity depends heavily on the assumption that the expected cell counts are at least moderately large; a minimum size of five is often quoted as a rule of thumb. Even when cell counts are adequate, the chi-square is only a large-sample approximation to the true distribution of chi-square under the null hypothesis. If the data set is smaller than is appropriate for a chi-square test, then Fisher's exact test may be preferable.

Performing Pearson's chi-square test

From the main menu, choose **Statistics ▶ Compare Samples ▶ Counts and Proportions ▶ Chi-square Test**. The **Pearson's Chi-Square Test** dialog opens, as shown in Figure 8.23.

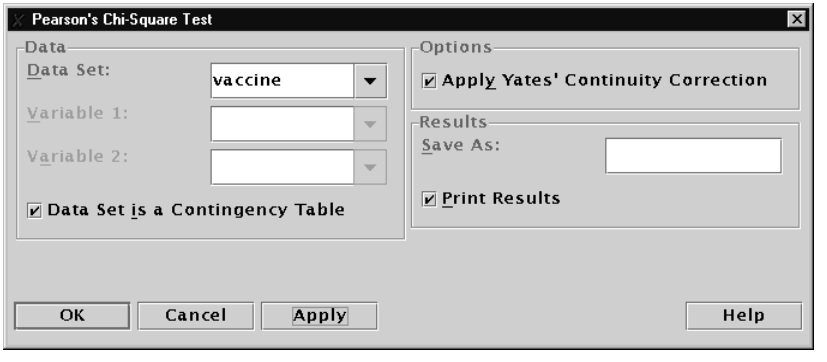


Figure 8.23: *The **Pearson's Chi-Square Test** dialog.*

Example

The data set shown in Table 8.8 contains a contingency table with results from Salk vaccine trials in the early 1950s. There are two categorical variables for the Salk trials: vaccination status, which has the two levels "vaccinated" and "placebo," and polio status, which has the three levels "no polio," "non-paralytic polio," and "paralytic polio." Of 200,745 individuals who were vaccinated, 24 contracted non-paralytic polio, 33 contracted paralytic polio, and the remaining 200,688 did not contract any kind of polio. Of 201,229 individuals

who received the placebo, 27 contracted non-paralytic polio, 115 contracted paralytic polio, and the remaining 201,087 did not contract any kind of polio.

Table 8.8: *A contingency table summarizing the results of the Salk vaccine trials.*

	None	Nonparalytic	Paralytic
Vaccinated	200,688	24	33
Placebo	201,087	27	115

When working with contingency table data, the primary interest is most often determining whether there is any association in the form of statistical dependence between the two categorical variables whose counts are displayed in the table. The null hypothesis is that the two variables are statistically independent.

Setting up the data

To create a `vaccine` data set containing the information in Table 8.8, type the following in the **Commands** window:

```
> vaccine <- data.frame(None = c(200688, 201087),
+ Nonparalytic=c(24,27), Paralytic=c(33,115),
+ row.names = c("Vaccinated", "Placebo"))
> vaccine
             None Nonparalytic Paralytic
Vaccinated 200688           24        33
   Placebo 201087           27       115
```

Statistical inference

We perform a chi-square test of independency for the `vaccine` data.

1. Open the **Pearson's Chi-Square Test** dialog.
2. Type `vaccine` in the **Data Set** field.
3. Select the **Data Set is a Contingency Table** check box, and click **OK**.

A summary of the test appears in the **Report** window. The p-value of 0 indicates that we reject the null hypothesis of independence. Vaccination and polio status are related.

POWER AND SAMPLE SIZE

When designing a study, one of the first questions to arise is how large a sample size is necessary. The sample size depends upon the minimum detectable difference of interest, the acceptable probability of rejecting a true null hypothesis (alpha), the desired probability of correctly rejecting a false null hypothesis (power), and the variability within the population(s) under study.

S-PLUS provides power and sample size calculations for one and two sample tests of normal means or binomial proportions.

- **Normal power and sample size:** computes sample sizes for statistics that are asymptotically normally distributed, such as a sample mean. Alternatively, it may be used to calculate power or minimum detectable difference for a sample of a specified size.
- **Binomial power and sample size:** computes sample sizes for statistics that are asymptotically binomially distributed, such as a proportion. Alternatively, it may be used to calculate power or minimum detectable difference for a sample of a specified size.

Normal Mean

The **Normal Power and Sample Size** dialog assists in computing sample sizes for statistics that are asymptotically normally distributed. Alternatively, it may be used to calculate power or minimum detectable difference for a sample of a specified size.

Computing power and sample size for a mean

From the main menu, choose **Statistics ▶ Power and Sample Size ▶ Normal Mean**. The **Normal Power and Sample Size** dialog opens, as shown in Figure 8.24.

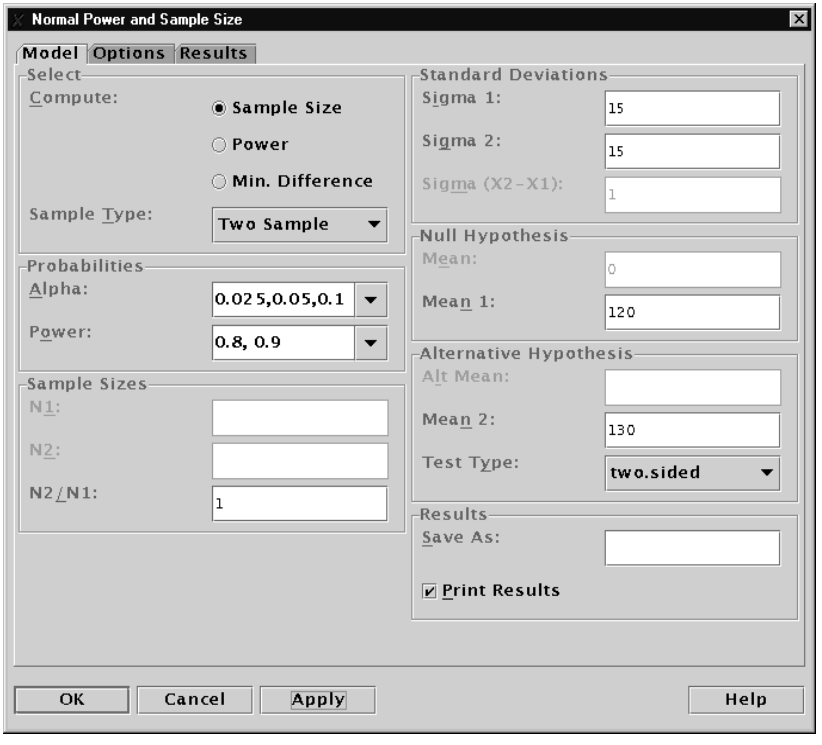


Figure 8.24: *The **Normal Power and Sample Size** dialog.*

Example

A scientist is exploring the efficacy of a new treatment. The plan is to apply the treatment to half of a study group, and then compare the levels of a diagnostic enzyme in the treatment subjects with the untreated control subjects. The scientist needs to determine how many subjects are needed in order to determine whether the treatment significantly changes the concentration of the diagnostic enzyme.

Historical information indicates that the average enzyme level is 120, with a standard deviation of 15. A difference in average level of 10 or more between the treatment and control groups is considered to be of clinical importance. The scientist wants to determine what sample

sizes are necessary for various combinations of alpha (the probability of falsely claiming the groups differ when they do not) and power (the probability of correctly claiming the groups differ when they do).

The **Normal Power and Sample Size** dialog produces a table of sample sizes for various combinations of alpha and power.

1. Open the **Normal Power and Sample Size** dialog.
2. Select `Two Sample` as the **Sample Type**.
3. Enter `120` as **Mean1**, `130` as **Mean2**, and `15` for both **Sigma1** and **Sigma2**.
4. Enter `0.025, 0.05, 0.1` for **Alpha**, and enter `0.8, 0.9` for **Power**. We calculate equal sample sizes for all combinations of these alpha and power values.
5. Click **OK**.

A power table is displayed in the **Report** window. The table indicates what sample sizes `n1` and `n2` are needed for each group at various levels of alpha and power. For example, the scientist needs 36 subjects per group to determine a difference of 10 at an alpha of 0.05 and power of 0.8.

```
   *** Power Table ***
  mean1 sd1 mean2 sd2 delta alpha power n1 n2
1   120  15   130  15    10 0.025   0.8 43 43
2   120  15   130  15    10 0.050   0.8 36 36
3   120  15   130  15    10 0.100   0.8 28 28
4   120  15   130  15    10 0.025   0.9 56 56
5   120  15   130  15    10 0.050   0.9 48 48
6   120  15   130  15    10 0.100   0.9 39 39
```

Binomial Proportion

The **Binomial Power and Sample Size** dialog assists in computing sample sizes for statistics that are asymptotically binomially distributed. Alternatively, it may be used to calculate power or minimum detectable difference for a sample of a specified size.

Computing power and sample size for a proportion

From the main menu, choose **Statistics ▶ Power and Sample Size ▶ Binomial Proportion**. The **Binomial Power and Sample Size** dialog opens, as shown in Figure 8.25.

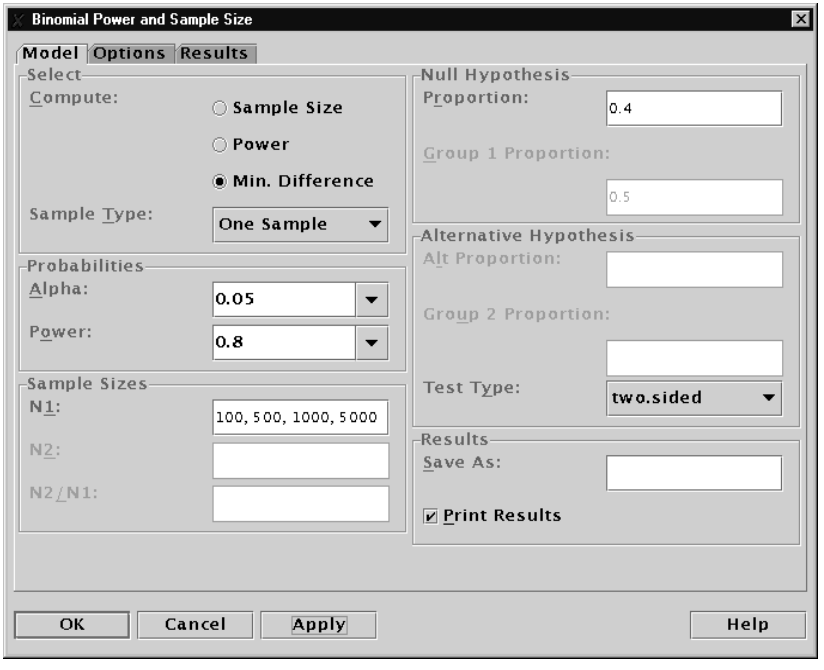


Figure 8.25: *The **Binomial Power and Sample Size** dialog.*

Example

Historically, 40% of the voters in a certain congressional district vote for the Democratic congressional candidate. A pollster is interested in determining the proportion of Democratic voters in an upcoming election. The pollster wants to know how sizable a difference could be detected for various sample sizes. That is, how much would the proportion of Democratic voters in the sample have to differ from the historical proportion of 40% to claim that the proportion is significantly different from the historical norm?

1. Open the **Binomial Power and Sample Size** dialog.
2. Select **Min. Difference** as the value to **Compute**. Enter `0.4` as the **Proportion** and `100, 500, 1000, 5000` as the sample sizes **N1** to consider.
3. Click **OK**.

A power table is displayed in the **Report** window. The table indicates the detectable differences `delta` for each sample size. For example, with 1000 observations the pollster could determine whether the proportion varies from 40% by at least 4.34%.

```
   *** Power Table ***
  p.null     p.alt     delta alpha power   n1
1    0.4 0.5372491 0.1372491  0.05   0.8  100
2    0.4 0.4613797 0.0613797  0.05   0.8  500
3    0.4 0.4434020 0.0434020  0.05   0.8 1000
4    0.4 0.4194100 0.0194100  0.05   0.8 5000
```

EXPERIMENTAL DESIGN

Typically, a researcher begins an experiment by generating a design, which is a data set indicating the combinations of experimental variables at which to take observations. The researcher then measures some outcome for the indicated combinations, and records this by adding a new column to the design data set. Once the outcome is recorded, exploratory plots may be used to examine the relationship between the outcome and the experimental variables. The data may then be analyzed using ANOVA or other techniques.

The **Factorial Design** and **Orthogonal Array Design** dialogs create experimental designs. The **Design Plot**, **Factor Plot**, and **Interaction Plot** dialogs produce exploratory plots for designs.

Factorial

The **Factorial Design** dialog creates a *factorial* or *fractional factorial* design. The basic factorial design contains all possible combinations of the variable levels, possibly replicated and randomized. A fractional factorial design excludes some combinations based upon which model effects are of interest.

Creating a factorial design

From the main menu, choose **Statistics ▶ Design ▶ Factorial**. The **Factorial Design** dialog opens, as shown in Figure 8.26.

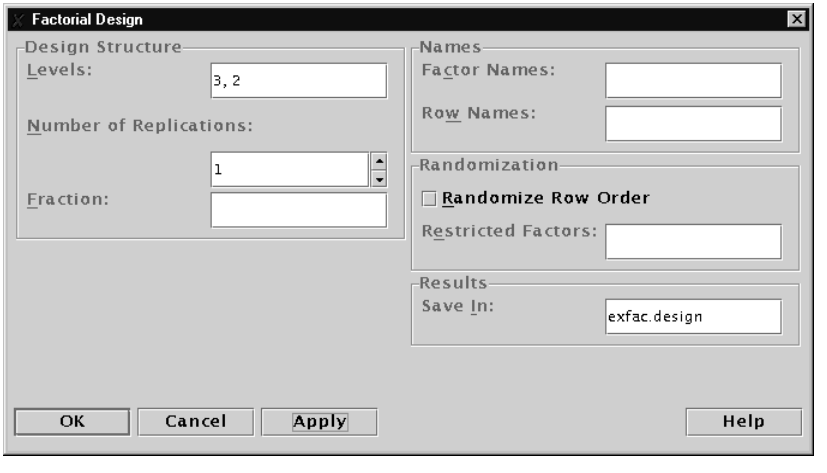


Figure 8.26: *The **Factorial Design** dialog.*

Example

We create a design with 3 levels of the first variable and two levels of the second.

1. Open the **Factorial Design** dialog.
2. Specify `3, 2` as the **Levels**.
3. Type `exfac.design` in the **Save In** field.
4. Click **OK**.

An `exfac.design` data set containing the design is created. You can view `exfac.design` with either the **Commands** window or the **Data** viewer.

Orthogonal Array

The **Orthogonal Array Design** dialog creates an orthogonal array design. *Orthogonal array designs* are essentially very sparse fractional factorial designs, constructed such that inferences may be made regarding main (first-order) effects. Level combinations necessary for estimating second- and higher-order effects are excluded in the interest of requiring as few measurements as possible.

Generating an orthogonal array design

From the main menu, choose **Statistics ▶ Design ▶ Orthogonal Array**. The **Orthogonal Array Design** dialog opens, as shown in Figure 8.27.

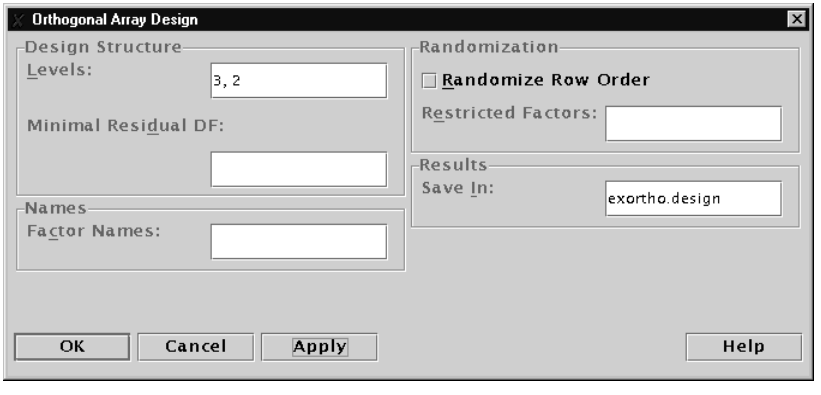


Figure 8.27: *The **Orthogonal Array Design** dialog.*

Example

We create a design with 3 levels of the first variable and two levels of the second.

1. Open the **Orthogonal Array Design** dialog.
2. Specify `3, 2` as the **Levels**.
3. Type `exortho.design` in the **Save In** field.
4. Click **OK**.

An `exortho.design` data set containing the design is created. You can view `exortho.design` with either the **Commands** window or the **Data** viewer. In this simple example, the orthogonal array design is equivalent to the design created in the section Factorial on page 327.

Design Plot

A *design plot* displays a function of a variable for each level of one or more corresponding factors. The default function is the mean.

Creating a design plot

From the main menu, choose **Statistics ▶ Design ▶ Design Plot**. The **Design Plot** dialog opens, as shown in Figure 8.28.

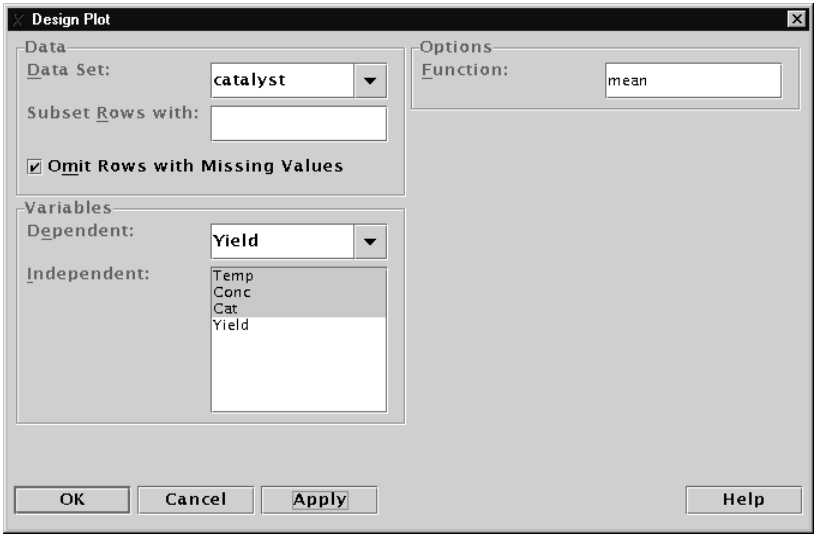


Figure 8.28: *The **Design Plot** dialog.*

Example

The `catalyst` data set comes from a designed experiment. Its eight rows represent all possible combinations of two temperatures (`Temp`), two concentrations (`Conc`), and two catalysts (`Cat`). The fourth column represents the response variable `Yield`. We are interested in determining how temperature, concentration, and catalyst affect the `Yield`. Prior to fitting an ANOVA model, we can use various plots to examine the relationship between these variables. We start with a design plot.

1. Open the **Design Plot** dialog.
2. Type `catalyst` in the **Data Set** field.
3. Select `Yield` as the **Dependent** variable.
4. CTRL-click to select `Temp`, `Conc`, and `Cat` as the **Independent** variables.
5. Click **OK**.

A design plot appears in a **Graph** window. This plot has a vertical bar for each factor, and a horizontal bar indicating the mean of `Yield` for each factor level.

Factor Plot

A *factor plot* consists of side by side plots comparing the values of a variable for different levels of a factor. By default, box plots are used. See the `plot.factor` help file for details.

Creating a factor plot

From the main menu, choose **Statistics ▶ Design ▶ Factor Plot**. The **Factor Plot** dialog opens, as shown in Figure 8.29.

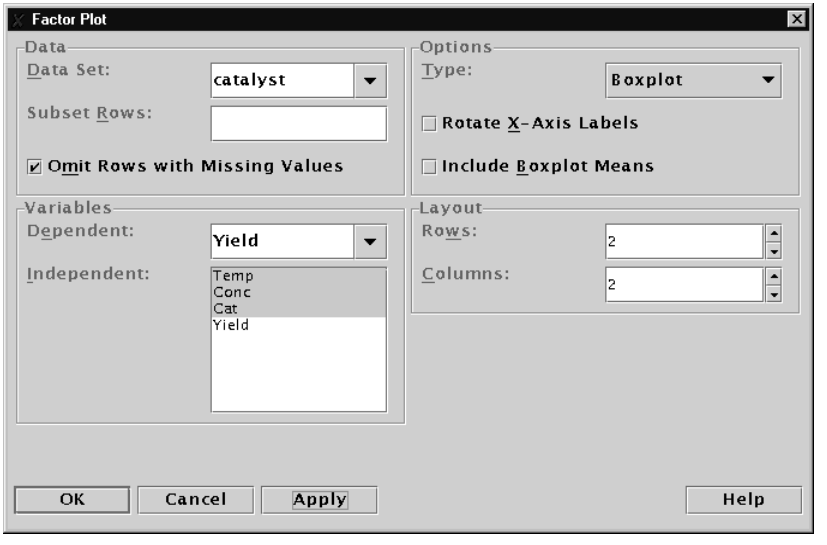


Figure 8.29: *The **Factor Plot** dialog.*

Example

We create factor plots for the `catalyst` data set as follows:

1. Open the **Factor Plot** dialog.
2. Type `catalyst` in the **Data Set** field.
3. Select `Yield` as the **Dependent** variable.
4. CTRL-click to select `Temp`, `Conc`, and `Cat` as the **Independent** variables.
5. Change the number of **Rows** and number of **Columns** to **2**. This specifies a 2×2 grid of plots.
6. Click **OK**.

A factor plot appears in a **Graph** window. For each factor there is a set of box plots for `Yield`, with a separate box plot for each factor level.

Interaction Plot

An *interaction plot* displays the levels of one factor along the x-axis, the response on the y-axis, and the points corresponding to a particular level of a second factor connected by lines. This type of plot is useful for exploring or discovering interactions.

Creating an interaction plot

From the main menu, choose **Statistics ▶ Design ▶ Interaction Plot**. The **Interaction Plot** dialog opens, as shown in Figure 8.30.

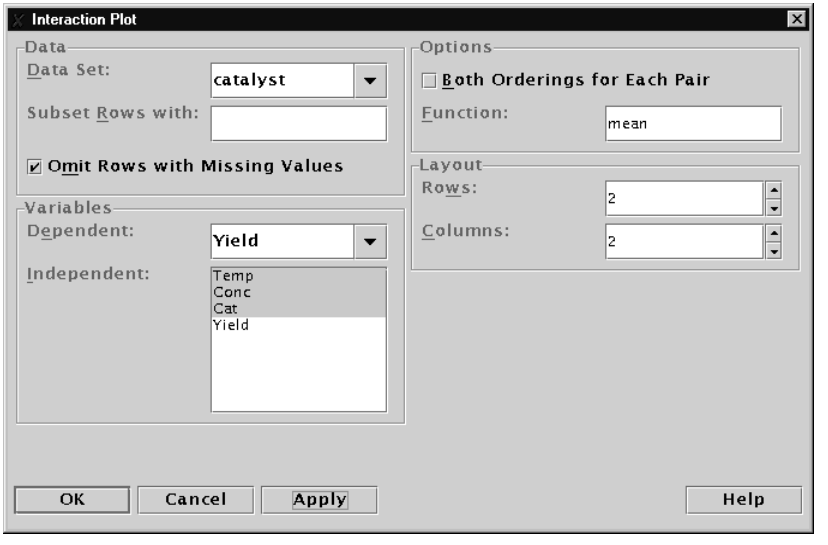


Figure 8.30: *The **Interaction Plot** dialog.*

Example

We create interaction plots for the `catalyst` data set as follows:

1. Open the **Interaction Plot** dialog.
2. Type `catalyst` in the **Data Set** field.
3. Select `Yield` as the **Dependent** variable
4. CTRL-click to select `Temp`, `Conc`, and `Cat` as the **Independent** variables.
5. Change the number of **Rows** and number of **Columns** to **2**. This specifies a 2×2 grid of plots.
6. Click **OK**.

An interaction plot appears in a **Graph** window. For each pair of factors, a set of lines is created showing the mean of `Yield` for each level of the second factor at each level of the first factor. If the lines in a plot cross, it suggests that an interaction is present between the two factors.

REGRESSION

Regression is the standard technique for assessing how various predictors relate to a response. This section discusses the regression techniques available from the **Statistics ► Regression** menu.

- **Linear regression**: predicting a continuous response as a linear function of predictors using a least-squares fitting criterion.
- **Robust MM regression**: predicting a continuous response using an MM based robust fitting criterion.
- **Robust LTS regression**: predicting a continuous response using a least-trimmed-squares fitting criterion.
- **Stepwise linear regression**: selecting which variables to employ in a linear regression model using a stepwise procedure.
- **Generalized additive models:** predicting a general response as a sum of nonparametric smooth univariate functions of the predictors.
- **Local (loess) regression**: predicting a continuous response as a nonparametric smooth function of the predictors using least- squares.
- **Nonlinear regression**: predicting a continuous response as a nonlinear function of the predictors using least-squares.
- **Generalized linear models:** predicting a general response as a linear combination of the predictors using maximum likelihood.
- **Log-linear (Poisson) regression**: predicting counts using Poisson maximum likelihood.

- **Logistic regression**: predicting a binary response using binomial maximum likelihood with a logistic link.
- **Probit regression:** predicting a binary response using binomial maximum likelihood with a probit link.

Linear Regression

Linear regression is used to describe the effect of continuous or categorical variables upon a continuous response. It is by far the most common regression procedure. The linear regression model assumes that the response is obtained by taking a specific linear combination of the predictors and adding random variation (error). The error is assumed to have a Gaussian (normal) distribution with constant variance, and to be independent of the predictor values.

Linear regression uses the method of *least-squares*, in which a line is fit that minimizes the sum of the squared residuals. Suppose a set of n observations of the response variable y_i correspond to a set of values of the predictor x_i according to the model $\hat{y} = f(\hat{x})$, where $\hat{y} = (y_1, y_2, \ldots, y_n)$ and $\hat{x} = (x_1, x_2, \ldots, x_n)$. The *i*th *residual* r_i is defined as the difference between the *i*th observation y_i and the *i*th fitted value $\hat{y}_i = \hat{f}(x_i)$: that is, $r_i = y_i - \hat{y}_i$. The method of least squares finds a set of fitted values that minimizes the sum $\sum_{i=1}^{n} r_i^2$.

If the response of interest is not continuous, then logistic regression, probit regression, log-linear regression, or generalized linear regression may be appropriate. If the predictors affect the response in a nonlinear way, then nonlinear regression, local regression, or generalized additive regression may be appropriate. If the data contain outliers or the errors are not Gaussian, then robust regression may be appropriate. If the focus is on the effect of categorical variables, then ANOVA may be appropriate. If the observations are correlated or random effects are present, then the mixed effect or generalized least squares model may be appropriate.

Other dialogs related to linear regression are **Stepwise Linear Regression**, **Compare Models**, and **Multiple Comparisons**. The **Stepwise Linear Regression** dialog uses a stepwise procedure to suggest which variables to include in a model. **Compare Models** provides tests for determining which of several models is most appropriate. **Multiple Comparisons** calculates effects for categorical predictors in linear regression or ANOVA.

Fitting a linear regression model

From the main menu, choose **Statistics ▶ Regression ▶ Linear**. The **Linear Regression** dialog opens, as shown in Figure 8.31.

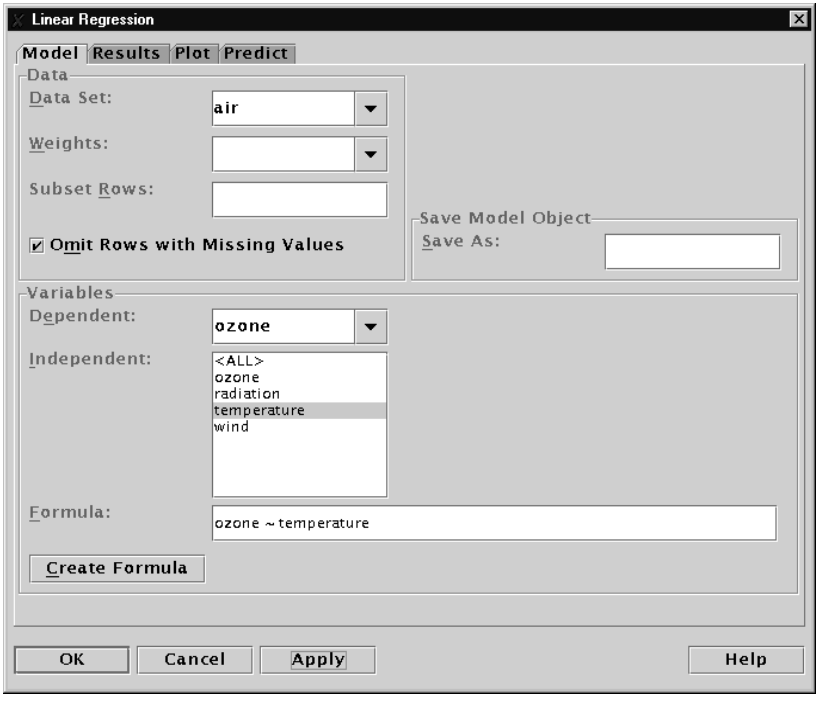


Figure 8.31: *The **Linear Regression** dialog.*

Example

We examine the air pollution data in the example data set `air`. This is a data set with 111 observations (rows) and 4 variables (columns). It is taken from an environmental study that measured the four variables

ozone, solar radiation, temperature, and wind speed for 111 consecutive days. We first create a scatter plot of the `temperature` and `ozone` variables in `air`, as shown in Figure 8.32.

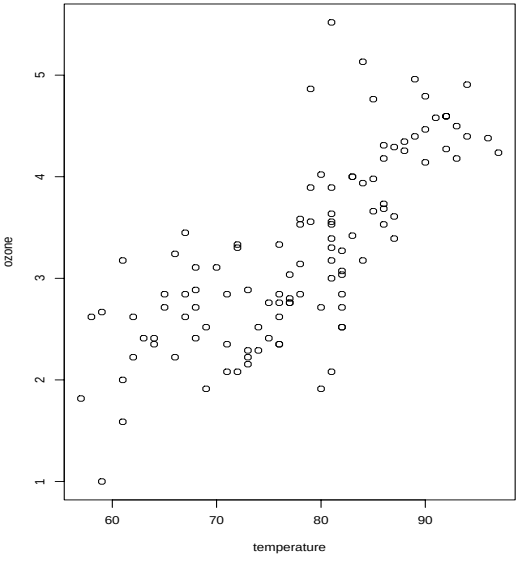


Figure 8.32: *A scatter plot of* `ozone` *versus* `temperature` .

From the scatter plot, we hypothesize a linear relationship between temperature and ozone concentration. We choose `ozone` as the response and `temperature` as the single predictor. The choice of response and predictor variables is driven by the subject matter in which the data arise, rather than by statistical considerations.

1. Open the **Linear Regression** dialog.
2. Type `air` in the **Data Set** field.
3. Type `ozone ~ temperature` in the **Formula** field. Alternatively, select `ozone` as the **Dependent** variable and `temperature` as the **Independent** variable. As a third way of generating a formula, click the **Create Formula** button and select `ozone` as the **Response** variable and `temperature` as a **Main Effect**. You can use the **Create Formula** button to

create complicated linear models and learn the notation for model specifications. The on-line help discusses formula creation in detail.

4. Go to the **Plot** page on the **Linear Regression** dialog and check the seven main diagnostic plots.
5. Click **OK** to do the linear regression.

S-PLUS generates a **Graph** window with seven diagnostic plots. You can access these plots by clicking the seven page tabs at the bottom of the **Graph** window. The plots appear similar to those shown in Figure 8.33. S-PLUS prints the results of the linear regression in the **Report** window:

```
        *** Linear Model ***
Call: lm(formula = ozone ~ temperature, data = air,
   na.action = na.exclude)
Residuals:
   Min      1Q  Median      3Q   Max
 -1.49 -0.4258 0.02521 0.3636 2.044

Coefficients:
               Value Std. Error  t value Pr(>|t|)
(Intercept)  -2.2260   0.4614    -4.8243   0.0000
temperature   0.0704   0.0059    11.9511   0.0000

Residual standard error: 0.5885 on 109 degrees of freedom
Multiple R-Squared: 0.5672
F-statistic: 142.8 on 1 and 109 degrees of freedom, the
  p-value is 0
```

The `Value` column under `Coefficients` gives the coefficients of the linear model, allowing us to read off the estimated regression line as follows:

$$\texttt{ozone} = -2.2260 + 0.0704 \times \texttt{temperature}$$

The column named `Std. Error` in the output gives the estimated standard error for each coefficient. The `Multiple R-Squared` term tells us that the model explains about 57% of the variation in `ozone`. The `F-statistic` is the ratio of the mean square of the regression to the estimated variance; if there is no relationship between

temperature and ozone, this ratio has an F distribution with 1 and 109 degrees of freedom. The ratio here is clearly significant, so the true slope of the regression line is probably not 0.

Diagnostic plots for linear models

How good is the fitted linear regression model? Is temperature an adequate predictor of ozone concentration? Can we do better? Questions such as these are essential any time you try to explain data with a statistical model. It is not enough to fit a model; you must also assess how well the model fits the data, and be prepared to modify the model or abandon it altogether if it does not satisfactorily explain the data.

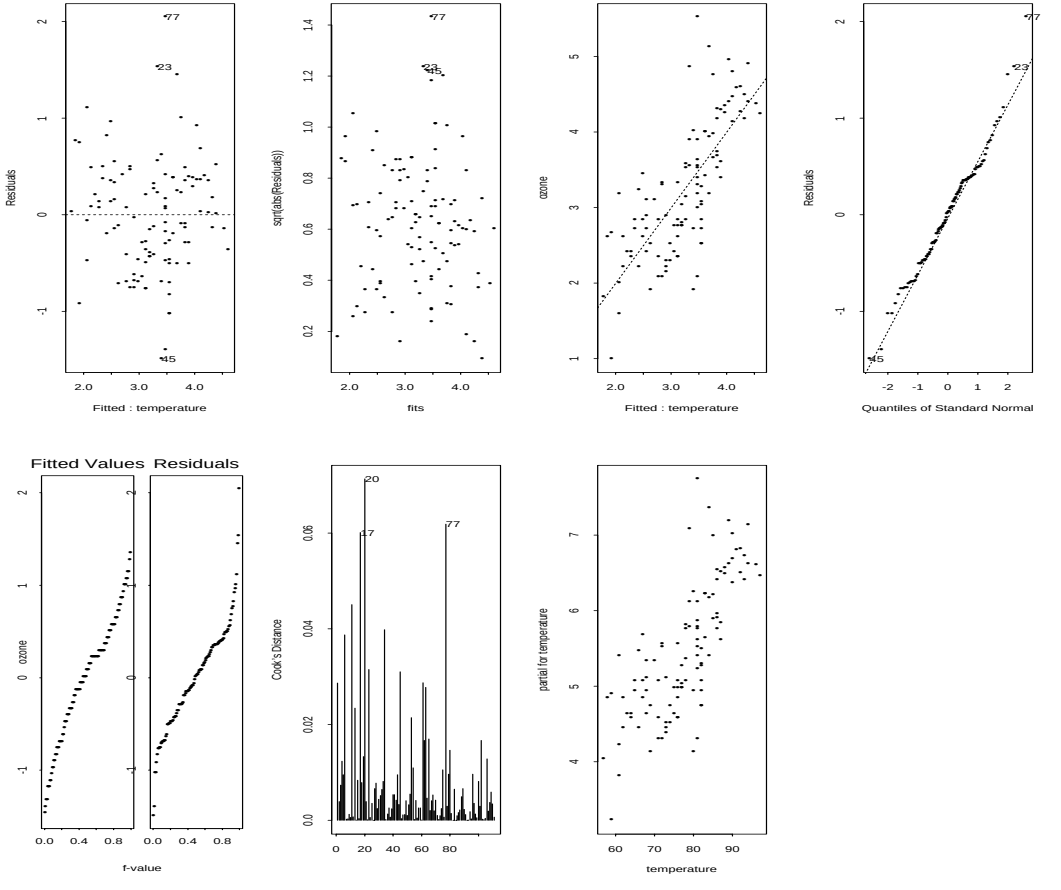


Figure 8.33: *Seven diagnostic plots created by the* ***Linear Regression*** *dialog.*

The simplest and most informative method for assessing the fit is to look at the model graphically, using an assortment of plots that, taken together, reveal the strengths and weaknesses of the model. For example, a plot of the response against the fitted values gives a good idea of how well the model has captured the broad outlines of the data. Examining a plot of the residuals against the fitted values often reveals unexplained structure left in the residuals, which should appear as nothing but noise in a strong model. The plotting options for the **Linear Regression** dialog provide these two plots, along with the following useful plots:

- *Square root of absolute residuals against fitted values.* This plot is useful in identifying outliers and visualizing structure in the residuals.
- *Normal quantile plot of residuals.* This plot provides a visual test of the assumption that the model's errors are normally distributed. If the ordered residuals cluster along the superimposed quantile-quantile line, you have strong evidence that the errors are indeed normal.
- *Residual-fit spread plot, or r-f plot.* This plot compares the spread of the fitted values with the spread of the residuals. Since the model is an attempt to explain the variation in the data, you hope that the spread in the fitted values is *much* greater than that in the residuals.
- *Cook's distance plot.* Cook's distance is a measure of the influence of individual observations on the regression coefficients.
- *Partial residual plot.* A partial residual plot is a plot of $r_i = b_k x_{ik}$ versus x_{ik}, where r_i is the ordinary residual for the ith observation, x_{ik} is the ith observation of the kth predictor, and b_k is the regression coefficient estimate for the kth predictor. Partial residual plots are useful for detecting nonlinearities and identifying possible causes of unduly large residuals.

The line $y = \hat{y}$ is shown as a dashed line in the third plot of the top row in Figure 8.33. In the case of simple regression, this line is visually equivalent to the regression line. The regression line appears

to model the trend of the data reasonably well. The residuals plots (left two plots in the top row of Figure 8.33) show no obvious pattern, although five observations appear to be outliers. By default, the three most extreme values are identified in each of the residuals plots and in the Cook's distance plot.

Another useful diagnostic plot is the normal plot of residuals (right plot in the top row of Figure 8.33). The normal plot gives no reason to doubt that the residuals are normally distributed. The r-f plot, on the other hand (left plot in the bottom row of Figure 8.33), shows a weakness in this model: the spread of the residuals is actually greater than the spread in the original data. However, if we ignore the five outlying residuals, the residuals are more tightly grouped than the original data.

The Cook's distance plot shows four or five heavily influential observations. Because the regression line fits the data reasonably well, the regression is significant, and the residuals appear normally distributed, we feel justified in using the regression line as a way to estimate the ozone concentration for a given temperature. One important issue remains, however: the regression line explains only 57% of the variation in the data. We may be able to do somewhat better by considering the effect of other variables on the ozone concentration.

Robust MM Regression

Robust regression models are useful for fitting linear relationships when the random variation in the data is not Gaussian (normal), or when the data contain significant outliers. In such situations, standard linear regression may return inaccurate estimates.

The *robust MM regression* method returns a model that is almost identical in structure to a standard linear regression model. This allows the production of familiar plots and summaries with a robust model. The MM method is the robust regression procedure currently recommended by MathSoft.

Performing robust MM regression

From the main menu, choose **Statistics ► Regression ► Robust MM**. The **Robust MM Linear Regression** dialog opens, as shown in Figure 8.34.

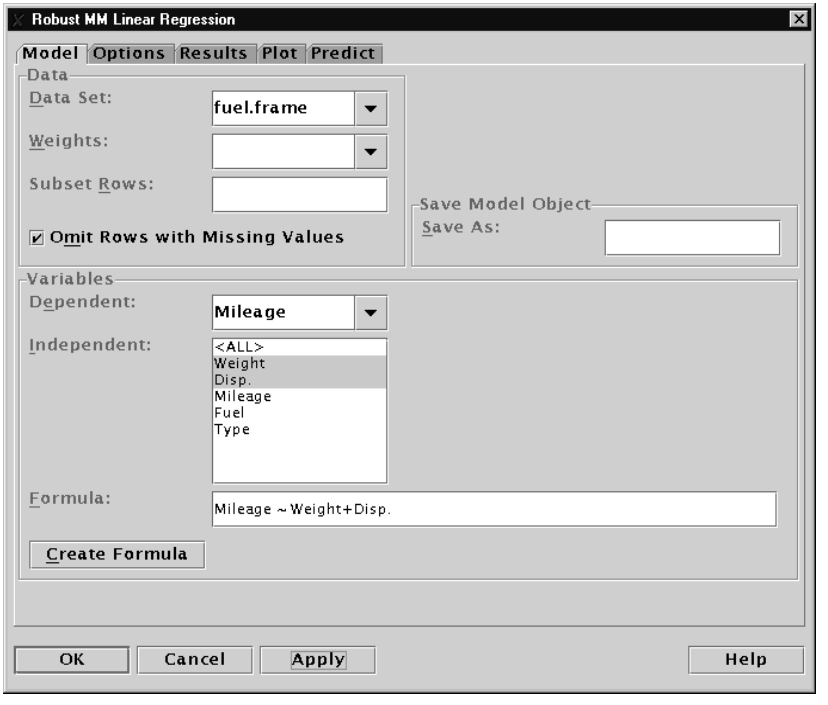


Figure 8.34: *The **Robust MM Linear Regression** dialog.*

Example

The data set `fuel.frame` is taken from the April 1990 issue of *Consumer Reports*. It contains 60 observations (rows) and 5 variables (columns). Observations of weight, engine displacement, mileage, type, and fuel were taken for each of sixty cars. In the `fuel.frame` data, we predict `Mileage` by `Weight` and `Disp.` using robust MM regression.

1. Open the **Robust MM Linear Regression** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. Type `Mileage~Weight+Disp.` in the **Formula** field. Alternatively, select `Mileage` as the **Dependent** variable and CTRL-click to select `Weight` and `Disp.` as the **Independent** variables. As a third way of generating a formula, click the **Create Formula** button, select `Mileage` as the **Response**

variable, and CTRL-click to select `Weight` and `Disp.` as the **Main Effect**s. You can use the **Create Formula** button to create complicated linear models and learn the notation for model specifications. The on-line help discusses formula creation in detail.

4. Click **OK** to fit the robust MM regression model.

A summary of the model appears in the **Report** window.

Robust LTS Regression

The *robust LTS regression* method performs least-trimmed-squares regression. It has less detailed plots and summaries than standard linear regression and robust MM regression.

Performing robust LTS regression

From the main menu, choose **Statistics ▶ Regression ▶ Robust LTS**. The **Robust LTS Linear Regression** dialog opens, as shown in Figure 8.35.

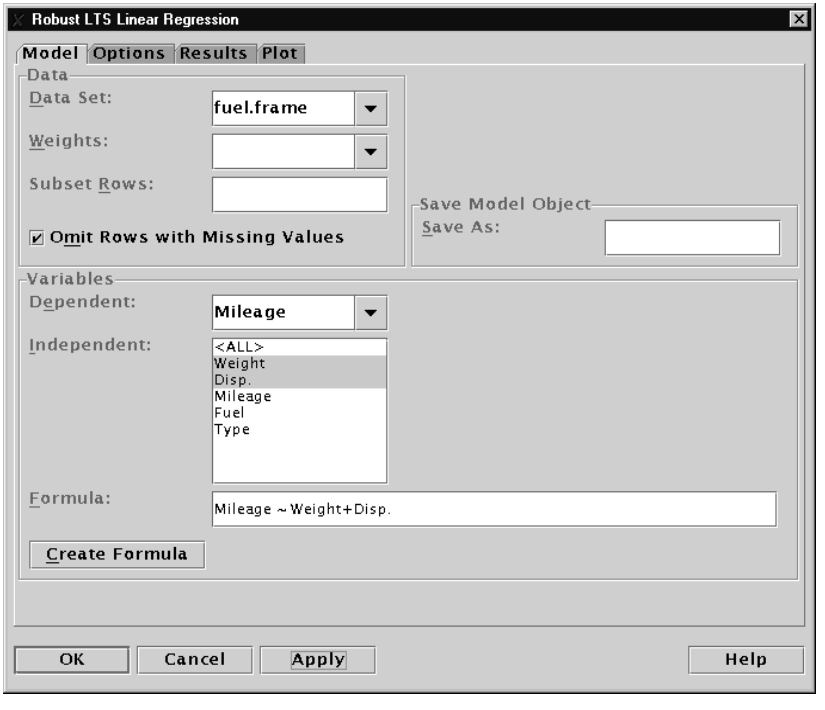


Figure 8.35: *The **Robust LTS Linear Regression** dialog.*

Example

In the `fuel.frame` data, we predict `Mileage` by `Weight` and `Disp.` using robust LTS regression.

1. Open the **Robust LTS Linear Regression** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. Type `Mileage~Weight+Disp.` in the **Formula** field. Alternatively, select `Mileage` as the **Dependent** variable and CTRL-click to select `Weight` and `Disp.` as the **Independent** variables. As a third way of generating a formula, click the **Create Formula** button, select `Mileage` as the **Response** variable, and CTRL-click to select `Weight` and `Disp.` as the **Main Effect**s. You can use the **Create Formula** button to create complicated linear models and learn the notation for model specifications. The on-line help discusses formula creation in detail.
4. Click **OK** to fit the robust LTS regression model.

A summary of the model appears in the **Report** window.

Stepwise Linear Regression

One step in the modeling process is determining what variables to include in the regression model. *Stepwise linear regression* is an automated procedure for selecting which variables to include in a regression model. Forward stepwise regression adds terms to the model until additional terms no longer improve the goodness-of-fit. At each step the term is added that most improves the fit. Backward stepwise regression drops terms from the model so long as dropping terms does not significantly decrease the goodness-of-fit. At each step the term is dropped whose removal least degrades the fit. Stepwise regression also has the option of alternating between adding and dropping terms. This is the default method used.

Performing stepwise linear regression

From the main menu, choose **Statistics ▶ Regression ▶ Stepwise**. The **Stepwise Linear Regression** dialog opens, as shown in Figure 8.36.

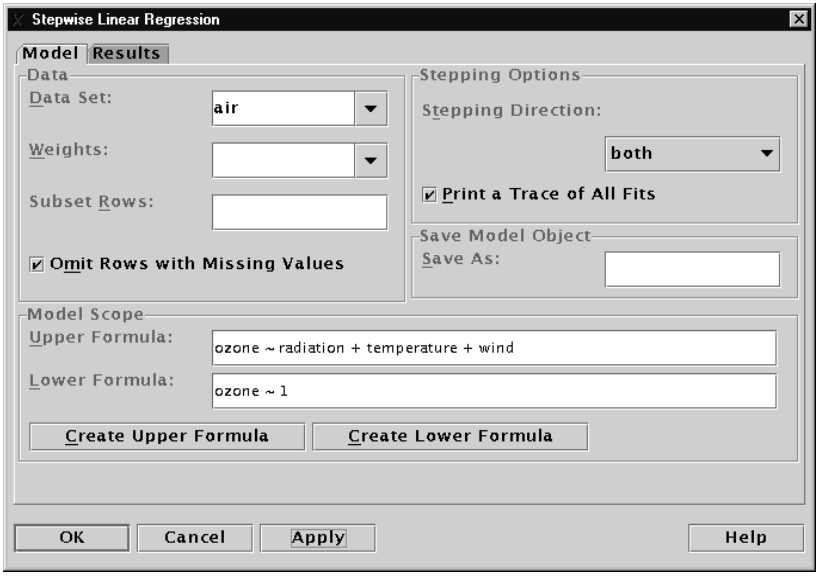


Figure 8.36: *The **Stepwise Linear Regression** dialog.*

Example

We apply stepwise regression to the `air` data.

1. Open the **Stepwise Linear Regression** dialog.
2. Type `air` in the **Data Set** field.
3. We must supply a formula representing the most complex model to consider. Specify `ozone ~ radiation + temperature + wind` as the **Upper Formula**.
4. We must also supply a formula representing the simplest model to consider. Specify `ozone ~ 1` as the **Lower Formula**. The 1 indicates inclusion of just an intercept term.
5. Click **OK**.

Stepwise regression uses the Cp statistic as a measure of goodness-of-fit. This is a statistic which rewards accuracy while penalizing model complexity. In this example, dropping any term yields a model with a Cp statistic that is smaller than that for the full model. Hence, the full model is selected as the best model.

The summary of the steps appears in the **Report** window.

```
  *** Stepwise Regression ***

  *** Stepwise Model Comparisons ***
Start:  AIC= 29.9302
 ozone ~ radiation + temperature + wind

Single term deletions

Model:
ozone ~ radiation + temperature + wind

scale:  0.2602624

            Df Sum of Sq      RSS       Cp
     <none>              27.84808 29.93018
  radiation  1   4.05928 31.90736 33.46893
temperature  1  17.48174 45.32982 46.89140
       wind  1   6.05985 33.90793 35.46950

  *** Linear Model ***

Call: lm(formula = ozone ~ radiation + temperature + wind,
data = air, na.action = na.exclude)
Residuals:
    Min      1Q   Median     3Q   Max
 -1.122 -0.3764 -0.02535 0.3361 1.495

Coefficients:
             Value Std. Error t value Pr(>|t|)
(Intercept) -0.2973  0.5552    -0.5355  0.5934
  radiation  0.0022  0.0006     3.9493  0.0001
temperature  0.0500  0.0061     8.1957  0.0000
       wind -0.0760  0.0158    -4.8253  0.0000

Residual standard error: 0.5102 on 107 degrees of freedom
Multiple R-Squared: 0.6807
F-statistic: 76.03 on 3 and 107 degrees of freedom, the
p-value is 0
```

Generalized Additive Models

Generalized additive models extend linear models by flexibly modeling additive nonlinear relationships between the predictors and the response. Whereas linear models assume that the response is linear in each predictor, additive models assume only that the response is affected by each predictor in a smooth way. The response is modeled as a sum of smooth functions in the predictors, where the smooth functions are estimated automatically using smoothers. Additive models may be useful for obtaining a final fit, or for exploring what types of variable transformations might be appropriate for use in a standard linear model.

Fitting an additive model

From the main menu, choose **Statistics ▶ Regression ▶ Generalized Additive**. The **Generalized Additive Models** dialog opens, as shown in Figure 8.37.

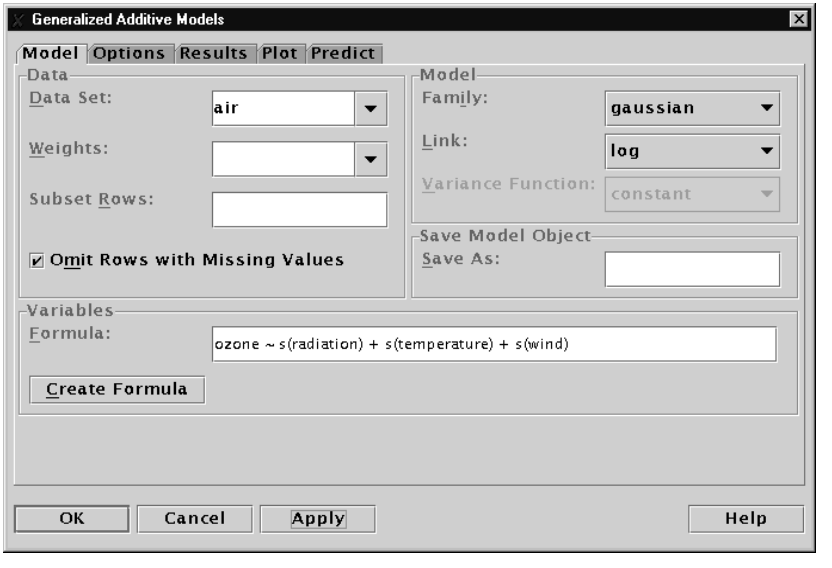


Figure 8.37: *The **Generalized Additive Models** dialog.*

Example

We fit an additive model for the `air` data.

1. Open the **Generalized Additive Models** dialog.
2. Type `air` in the **Data Set** field.

3. Specify `ozone ~ s(radiation) + s(temperature) + s(wind)` as the **Formula**.
4. On the **Plot** page of the dialog, select the **Partial Residuals** and **Include Partial Fits** check boxes. This indicates that we want plots of the partial residuals and partial fits for each predictor.
5. Click **OK**.

A summary of the additive model appears in the **Report** window. A multipage **Graph** window appears with one partial residual plot on each page.

Local (Loess) Regression

Local regression is a nonparametric generalization of multivariate polynomial regression. It is best thought of as a way to fit general smooth surfaces. A wide variety of options are available for specifying the form of the surface.

Fitting a local regression

From the main menu, choose **Statistics ► Regression ► Local (Loess)**. The **Local (Loess) Regression** dialog opens, as shown in Figure 8.38.

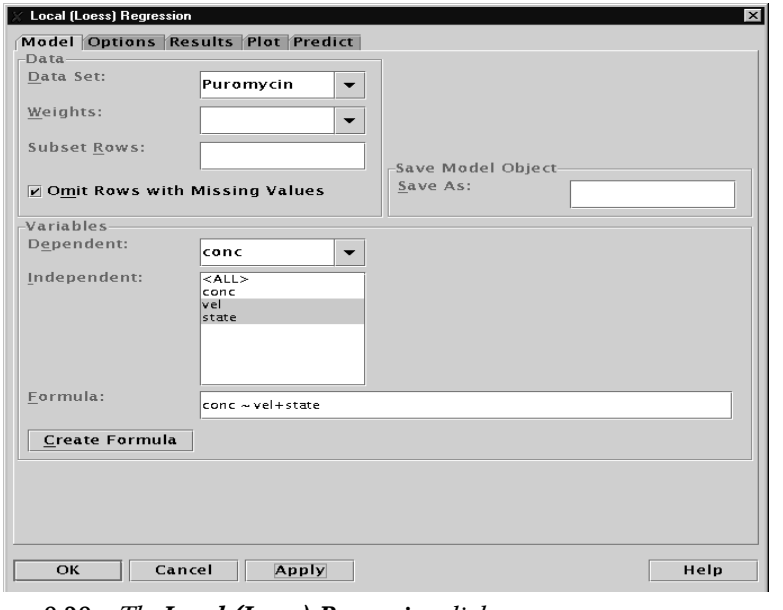


Figure 8.38: *The **Local (Loess) Regression** dialog.*

Example

The data set `Puromycin` has 23 rows representing the measurement of initial velocity of a biochemical reaction for 6 different concentrations of substrate and two different cell treatments. The section Nonlinear Regression describes these data in detail and discusses a theoretical model for the data. Before fitting a theoretical model, we can use the **Local (Loess) Regression** dialog to fit nonparametric smooth curves to the data.

Our model consists of a separate curve for each treatment group. We predict the response `conc` by the variables `vel` and `state`. Since `state` is a factor, this fits a separate smooth curve in `vel` for each level of `state`.

1. Open the **Local (Loess) Regression** dialog.
2. Type `Puromycin` in the **Data Set** field.
3. Type `conc~vel+state` in the **Formula** field. Alternatively, select `conc` as the **Dependent** variable and CTRL-click to select `vel` and `state` as the **Independent** variables. As a third way of generating a formula, click the **Create Formula** button, select `conc` as the **Response** variable, and CTRL-click to select `vel` and `state` as the **Main Effect**s. You can use the **Create Formula** button to create complicated linear models and learn the notation for model specifications. The on-line help discusses formula creation in detail.
4. On the **Plot** page of the dialog, select **Cond. Plots of Fitted vs Predictors**. This type of plot displays a separate plot in one variable for different subsets of another variable. In our case, it plots a separate curve for each level of `state`.
5. Click **OK**.

A summary of the loess model is presented in the **Report** window, and a **Graph** window displays the conditional plot.

Nonlinear Regression

Nonlinear regression uses a specific nonlinear relationship to predict a continuous variable from one or more predictor variables. The form of the nonlinear relationship is usually derived from an application-specific theoretical model.

The **Nonlinear Regression** dialog fits a nonlinear regression model. To use nonlinear regression, specify the form of the model in S-PLUS syntax and provide starting values for the parameter estimates.

Fitting a nonlinear least squares regression

From the main menu, choose **Statistics ► Regression ► Nonlinear**. The **Nonlinear Regression** dialog opens, as shown in Figure 8.39.

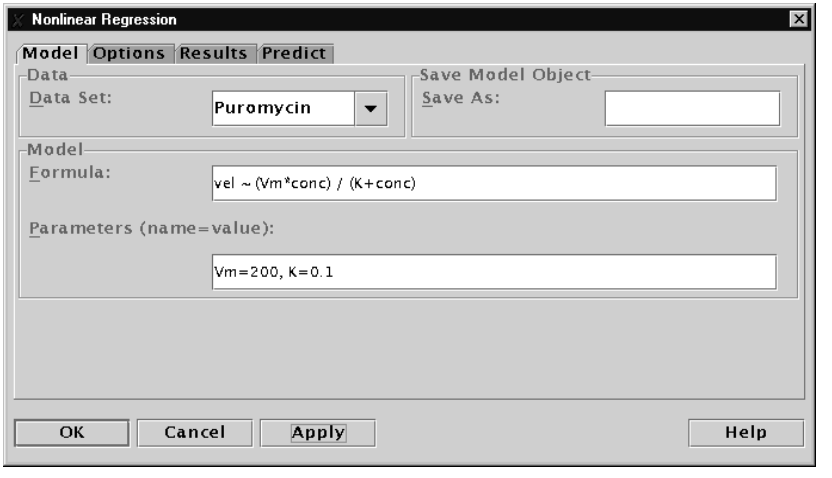


Figure 8.39: *The **Nonlinear Regression** dialog.*

Example

The data set `Puromycin` has 23 rows representing the measurement of initial velocity of a biochemical reaction for 6 different concentrations of substrate and two different cell treatments. Figure 8.40 plots velocity versus concentration with different symbols for the two treatment groups (treated and untreated).

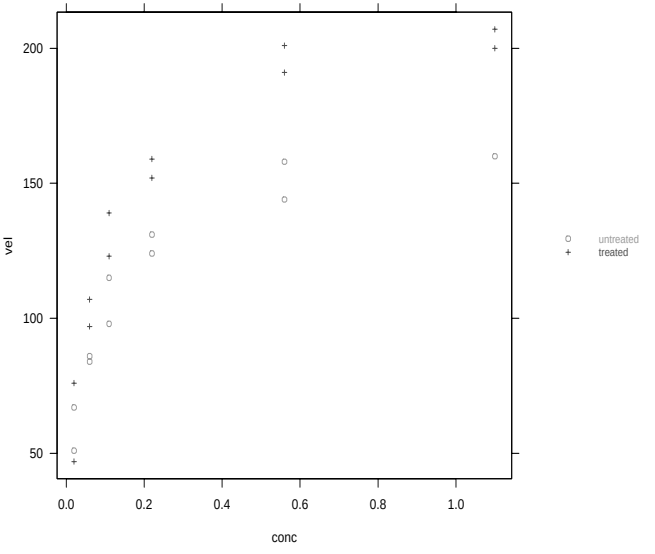


Figure 8.40: *Scatter plot of the* `Puromycin` *data.*

The relationship between velocity and concentration is known to follow a Michaelis-Menten relationship:

$$V = \frac{V_{max} c}{K + c} + \varepsilon$$

where V is the velocity, c is the enzyme concentration, V_{max} is a parameter representing the asymptotic velocity as $c \to \infty$, K is the Michaelis parameter, and ε is experimental error. Assuming the treatment with the drug would change V_{max} but not K, the optimization function is:

$$S(V_{max}, K) = \sum \left(V_i - \frac{(V_{max} + \Delta V_{max} I_{\{treated\}}(state)) c_i}{K + c_i} \right)^2$$

where $I_{\{treated\}}$ is the function indicating whether the cell was treated with Puromycin.

We first fit the simpler model in which a single curve is fit for both groups. We then add a term reflecting the influence of treatment.

In order to fit a nonlinear regression model, we must specify the form of the nonlinear model, the name of the data set, and starting values for the parameter estimates. Examination of Figure 8.40 suggests starting values of `V=200` and `K=0.1`, treating all observations as a single group. We fit a Michaelis-Menten relationship between velocity and concentration as follows:

1. Open the **Nonlinear Regression** dialog.
2. Type `Puromycin` in the **Data Set** field.
3. Type the Michaelis-Menten relationship `vel~(Vm*conc)/(K+conc)` into the **Formula** field.
4. Type the parameter starting values `Vm=200, K=0.1` into the **Parameters** field.
5. Click **OK**.

The following results appear in the **Report** window.

```
   *** Nonlinear Regression Model ***

 Formula: vel ~ (Vm * conc)/(K + conc)

 Parameters:
         Value Std. Error  t value
Vm 190.8050000  8.7644700 21.77030
 K   0.0603863  0.0107682  5.60785

 Residual standard error: 18.6146 on 21 degrees of freedom

 Correlation of Parameter Estimates:
     Vm
K 0.776
```

The printed results provide parameter estimates, standard errors, and t-values, as well as the residual standard error and correlation of parameter estimates.

We now fit a model containing a treatment effect:

1. Open the **Nonlinear Regression** dialog.
2. Type `Puromycin` in the **Data Set** field.
3. Type the Michaelis-Menten relationship `vel ~ ((Vm+delV * (state == "treated")) * conc)/(K + conc)` into the **Formula** field.
4. Figure 8.40 suggests starting values of `Vm=160` and `delV=40`, while the previous model suggests `K=0.05`. Type the starting values `Vm=160, delV=40, K=0.05` into the **Parameters** field.
5. Click **OK**.

The following results appear in the **Report** window.

```
   *** Nonlinear Regression Model ***

Formula: vel ~ ((Vm + delV * (state == "treated")) * conc)/
(K + conc)

Parameters:
           Value Std. Error  t value
  Vm 166.6010000 5.80726000 28.68840
delV  42.0245000 6.27201000  6.70032
   K   0.0579659 0.00590968  9.80863

Residual standard error: 10.5851 on 20 degrees of freedom

Correlation of Parameter Estimates:
          Vm    delV
delV -0.5410
   K  0.6110  0.0644
```

The printed results provide parameter estimates, standard errors, and t-values, as well as the residual standard error and correlation of parameter estimates. The magnitude of the t-statistic for `delV` confirms that the treatment affects the maximum velocity.

Generalized Linear Models

Generalized linear models are generalizations of the familiar linear regression model to situations where the response is discrete or the model varies in other ways from the standard linear model. The most widely used generalized linear models are logistic regression models for binary data and log-linear (Poisson) models for count data.

Fitting a generalized linear model

From the main menu, choose **Statistics ▶ Regression ▶ Generalized Linear**. The **Generalized Linear Models** dialog opens, as shown in Figure 8.41.

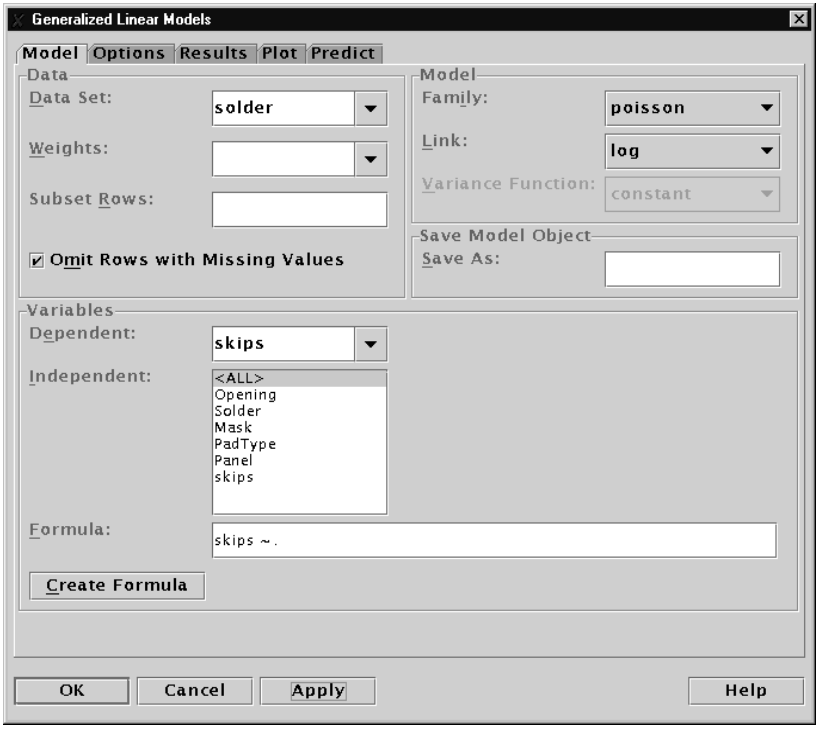


Figure 8.41: *The **Generalized Linear Models** dialog.*

Example

The `solder` data set contains 900 observations (rows) that are the results of an experiment that varied five factors relevant to the wave-soldering procedure for mounting components on printed circuit boards. The response variable `skips` is a count of how many solder

skips appeared in a visual inspection. We can use the **Generalized Linear Models** dialog to assess which process variables affect the number of skips.

1. Open the **Generalized Linear Models** dialog.
2. Type `solder` in the **Data Set** field.
3. Select `skips` as the **Dependent** variable and **<ALL>** in the **Independent** variable list. This generates `skips ~ .` in the **Formula** field.
4. Select `poisson` as the **Family**. The **Link** changes to `log`, which is the canonical link for a Poisson model.
5. Click **OK**.

A summary of the Poisson regression appears in the **Report** window.

Log-Linear (Poisson) Regression

Count data are frequently modeled using *log-linear regression*. In log-linear regression, the response is assumed to be generated from a Poisson distribution, with a centrality parameter that depends upon the values of the covariates.

Fitting a log-linear (Poisson) regression

From the main menu, choose **Statistics ▶ Regression ▶ Log-linear (Poisson)**. The **Log-linear (Poisson) Regression** dialog opens, as shown in Figure 8.42.

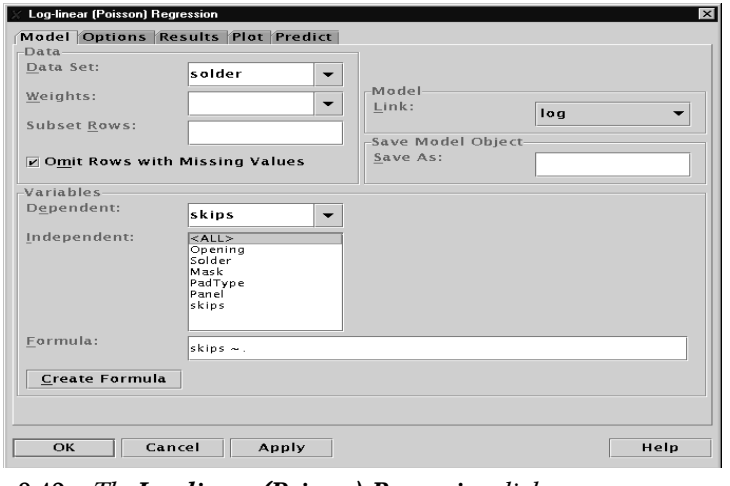


Figure 8.42: *The **Log-linear (Poisson) Regression** dialog.*

Example

In this example, we fit a Poisson regression to the `solder` data.

1. Open the **Log-linear** (**Poisson**) **Regression** dialog.
2. Type `solder` in the **Data Set** field.
3. Select `skips` as the **Dependent** variable and **<ALL>** in the **Independent** variable list. This generates `skips ~ .` in the **Formula** field.
4. Click **OK**.

A summary of the log-linear regression appears in the **Report** window. The t-values in the resulting table of coefficients are all fairly large, indicating that all of the process variables have a significant influence upon the number of skips generated.

Logistic Regression

Logistic regression models the relationship between a dichotomous response variable and one or more predictor variables. A linear combination of the predictor variables is found using maximum likelihood estimation, where the response variable is assumed to be generated by a binomial process whose probability parameter depends upon the values of the predictor variables.

Fitting a logistic regression

From the main menu, choose **Statistics ▶ Regression ▶ Logistic**. The **Logistic Regression** dialog opens, as shown in Figure 8.43.

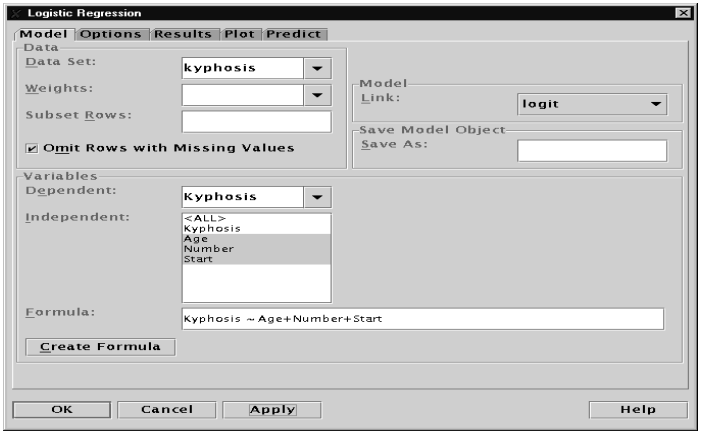


Figure 8.43: *The **Logistic Regression** dialog.*

Example

The data set `kyphosis` has 81 rows representing data on 81 children who have had corrective spinal surgery. The outcome `Kyphosis` is a binary variable, and the other three variables `Age`, `Number`, and `Start`, are numeric. Figure 8.44 displays box plots of `Age`, `Number`, and `Start` for each level of `Kyphosis`, as generated by the following commands:

```
> par(mfrow=c(3,1))
> boxplot(split(kyphosis$Age, kyphosis$Kyphosis),
+ xlab="Kyphosis", ylab="Age")
> boxplot(split(kyphosis$Number, kyphosis$Kyphosis),
+ xlab="Kyphosis", ylab="Number")
> boxplot(split(kyphosis$Start, kyphosis$Kyphosis),
+ xlab="Kyphosis", ylab="Start")
```

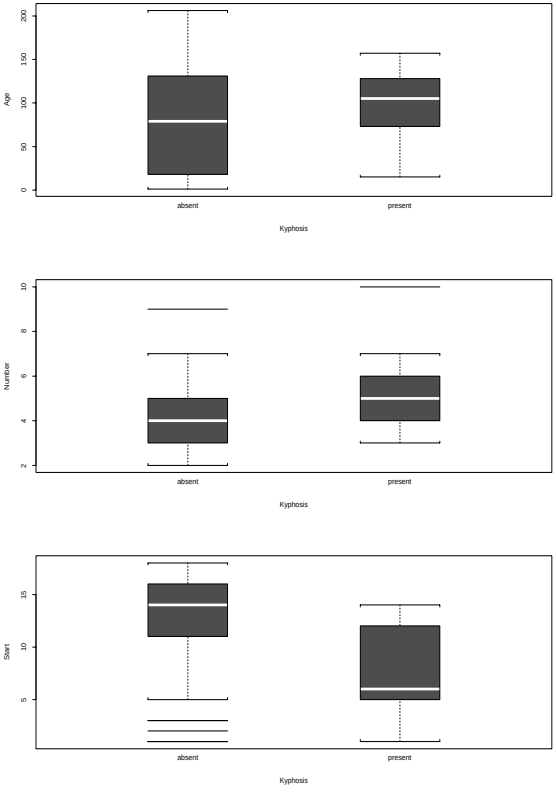


Figure 8.44: *Box plots of the Kyphosis data.*

Kyphosis is a postoperative spinal deformity. We are interested in exploring how the covariates influence whether or not the deformity occurs. Both `Start` and `Number` show strong location shifts with respect to the presence or absence of `Kyphosis`. The `Age` variable does not show such a shift in location. We can use logistic regression to quantify the influence of each covariate upon the likelihood of deformity.

1. Open the **Logistic Regression** dialog.
2. Type `kyphosis` in the **Data Set** field.
3. Specify `Kyphosis~Age+Number+Start` in the **Formula** field.
4. Click **OK**.

A summary of the logistic regression appears in the **Report** window. The summary contains information on the residuals, coefficients, and deviance. The high t-value for `Start` indicates it has a significant influence upon whether kyphosis occurs. The t-values for `Age` and `Number` are not large enough to display a significant influence upon the response.

```
   *** Generalized Linear Model ***

 Call: glm(formula = Kyphosis ~ Age + Number + Start,
   family = binomial(link = logit), data = kyphosis,
   na.action = na.exclude, control = list(
   epsilon = 0.0001, maxit = 50, trace = F))

 Deviance Residuals:
        Min          1Q      Median          3Q     Max
  -2.312363 -0.5484308 -0.3631876 -0.1658653 2.16133

 Coefficients:
                   Value Std. Error   t value
 (Intercept) -2.03693225 1.44918287 -1.405573
         Age  0.01093048 0.00644419  1.696175
      Number  0.41060098 0.22478659  1.826626
       Start -0.20651000 0.06768504 -3.051043
```

```
(Dispersion Parameter for Binomial family taken to be 1 )

    Null Deviance: 83.23447 on 80 degrees of freedom

Residual Deviance: 61.37993 on 77 degrees of freedom

Number of Fisher Scoring Iterations: 5
```

Probit Regression

The **Probit Regression** dialog fits a probit response model. This is a variation of logistic regression suitable for binomial response data.

Fitting a probit regression model

From the main menu, choose **Statistics ▶ Regression ▶ Probit**. The **Probit Regression** dialog opens, as shown in Figure 8.45.

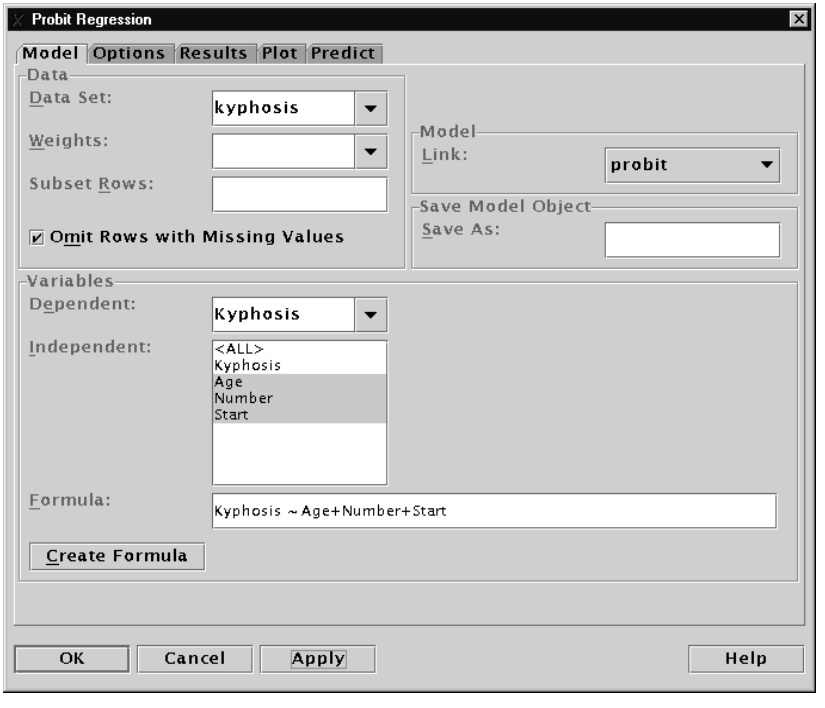


Figure 8.45: *The **Probit Regression** dialog.*

Example

In this example, we fit a probit regression model to the `kyphosis` data set:

1. Open the **Probit Regression** dialog.
2. Type `kyphosis` in the **Data Set** field.
3. Specify `Kyphosis~Age+Number+Start` in the **Formula** field.
4. Click **OK**.

A summary of the model is printed in the **Report** window.

```
  *** Generalized Linear Model ***

Call: glm(formula = Kyphosis ~ Age + Number + Start,
  family = binomial(link = probit), data = kyphosis,
  na.action = na.exclude, control = list(epsilon =
  0.0001, maxit = 50, trace = F))
Deviance Residuals:
       Min         1Q     Median         3Q      Max
 -2.217301 -0.5440968 -0.3535132 -0.124005 2.149486

Coefficients:
                  Value  Std. Error   t value
(Intercept) -1.063353291 0.809886949 -1.312965
        Age  0.005984768 0.003507093  1.706475
     Number  0.215179016 0.121687912  1.768286
      Start -0.120214682 0.038512786 -3.121423

(Dispersion Parameter for Binomial family taken to be 1 )

    Null Deviance: 83.23447 on 80 degrees of freedom

Residual Deviance: 61.0795 on 77 degrees of freedom

Number of Fisher Scoring Iterations: 5
```

ANALYSIS OF VARIANCE

Analysis of variance (ANOVA) is generally used to explore the influence of one or more categorical variables upon a continuous response.

Fixed Effects ANOVA

The **ANOVA** dialog performs classical fixed effects analysis of variance.

Fitting a fixed effects ANOVA model

From the main menu, choose **Statistics ► ANOVA ► Fixed Effects**. The **ANOVA** dialog opens, as shown in Figure 8.46.

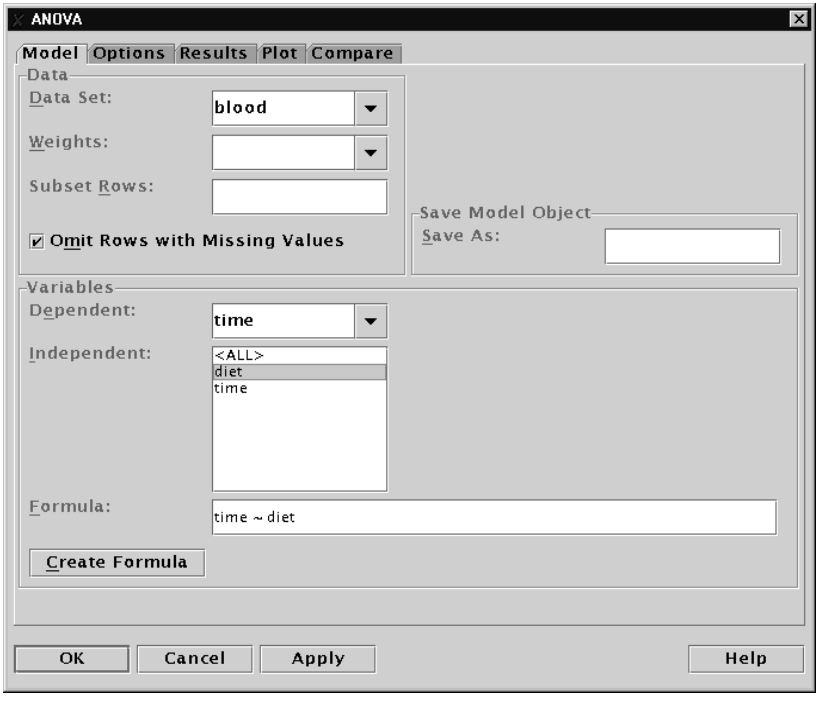


Figure 8.46: *The **ANOVA** dialog.*

Example

In the section One-Way Analysis of Variance on page 298, we performed a simple one-way ANOVA on the `blood` data set listed in Table 8.2. These data give the blood coagulation times for four different diets. In general, the **ANOVA** dialog can handle far more complicated designs than the one-way ANOVA dialog. In addition, it generates diagnostic plots and provides more information on the results of the analysis. We use the **ANOVA** dialog to reproduce the results of the earlier example. We also generate some diagnostic plots to see how well our model suits our data.

1. If you have not done so already, create the `blood` data set with the instructions given on page 300.
2. Open the **ANOVA** dialog.
3. Enter `blood` as the **Data Set**.
4. Enter the formula `time ~ diet` for the one-way ANOVA we are going to perform. Alternatively, select `time` as the **Dependent** variable and `diet` as the **Independent** variable. As a third way of generating a formula, click the **Create Formula** button, select `time` as the **Response** variable and `diet` as a **Main Effect**. You can use the **Create Formula** button to create complicated linear models and learn the notation for model specifications. The on-line help discusses formula creation in detail.
5. Click on the **Plot** page and check all seven possible plots.
6. Click **OK** to do the analysis.

S-PLUS generates seven diagnostic plots. You can access these plots by clicking the seven page tabs at the bottom of the **Graph** window. The plots do not reveal any significant problems in our model. The **Report** window displays the results of the ANOVA.

Random Effects ANOVA

Random effects ANOVA is used in balanced designed experiments where the treatment effects are taken to be random. The model must be balanced, and the model must be fully random. Only single strata designs are allowed.

For mixed effect models, use the **Linear Mixed Effects** dialog.

Fitting a random effects ANOVA model

From the main menu, choose **Statistics ► ANOVA ► Random Effects**. The **Random Effects Analysis of Variance** dialog opens, as shown in Figure 8.47.

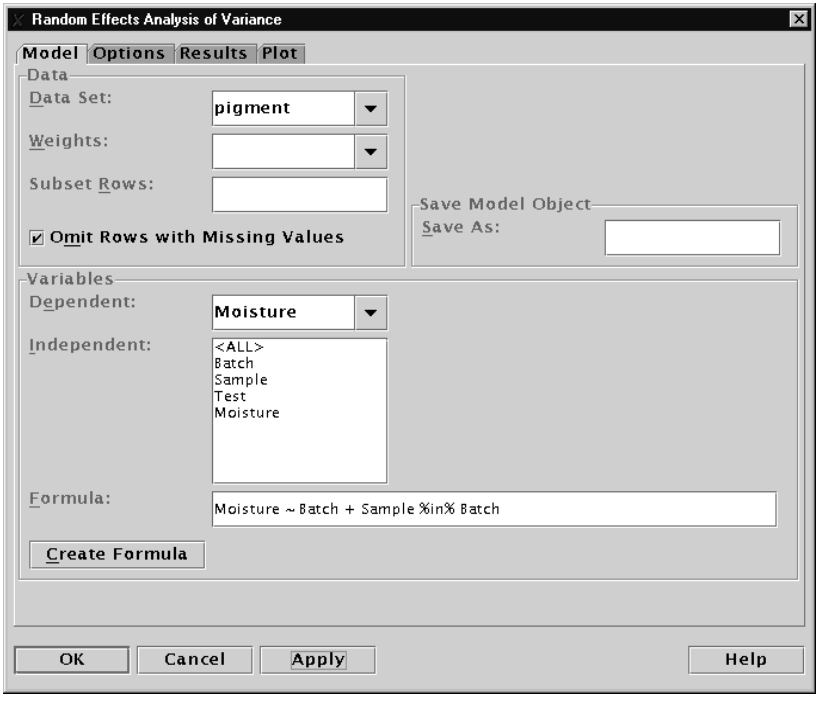


Figure 8.47: *The **Random Effects Analysis of Variance** dialog.*

Example

The `pigment` data set has 60 rows and 4 columns. The rows represent 15 batches of pigment for which 2 samples were drawn from each batch, and 2 analyses were made on each sample. These data are from a designed experiment of moisture content where samples are nested within batch. We fit a random effects ANOVA model to assess the within-batch and between-batch variation.

1. Open the **Random Effects Analysis of Variance** dialog.
2. Type `pigment` in the **Data Set** field.

3. Enter the following **Formula**:

```
Moisture ~ Batch + Sample %in% Batch
```

4. Click **OK**.

A summary of the model is printed in the **Report** window.

Multiple Comparisons

Analysis of variance models are typically used to compare the effects of several treatments upon some response. After an analysis of variance model has been fit, it is often of interest to determine whether any significant differences exist between the responses for the various treatment groups and, if so, to estimate the size of the differences. *Multiple comparisons* provides tests for equality of effects and also estimates treatment effects.

The **Multiple Comparisons** dialog calculates simultaneous or nonsimultaneous confidence intervals for any number of estimable linear combinations of the parameters of a fixed-effects linear model. It requires the name of an analysis of variance model (`aov`) or linear model (`lm`), and specification of which effects are of interest.

The Multiple Comparisons functionality is also available on the **Compare** page of the **ANOVA** dialog.

Performing multiple comparisons

From the main menu, choose **Statistics ► ANOVA ► Multiple Comparisons**. The **Multiple Comparisons** dialog opens, as shown in Figure 8.48.

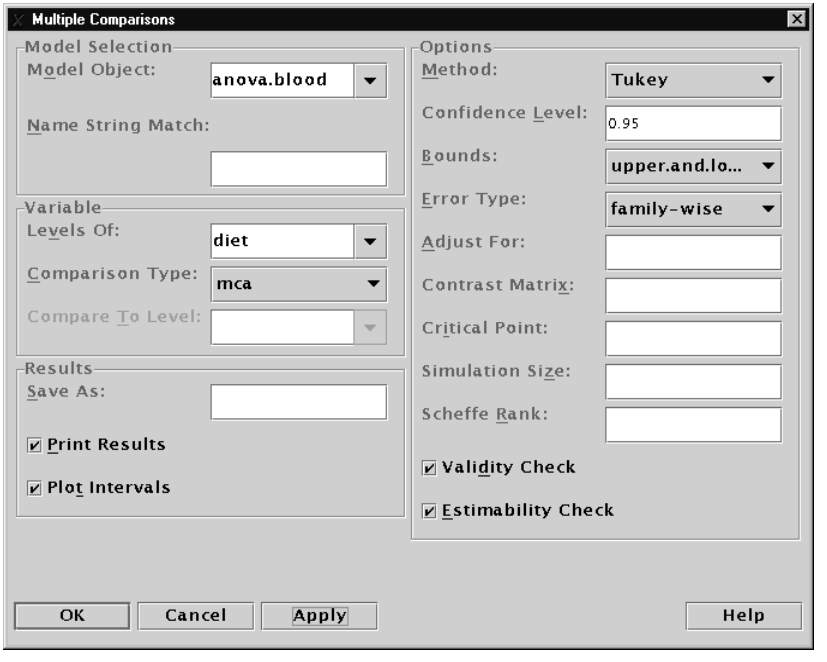


Figure 8.48: *The **Multiple Comparisons** dialog.*

Example

In the section One-Way Analysis of Variance on page 298, we performed a simple one-way ANOVA on the `blood` data set listed in Table 8.2. These data give the blood coagulation times for four different diets. In the section Fixed Effects ANOVA on page 361**,** we revisited the `blood` data set and concluded that diet affects blood coagulation times. The next step is to generate multiple simultaneous confidence intervals to see which diets are different from each other. We can do this using either the **Compare** page on the **ANOVA** dialog or the **Multiple Comparisons** dialog.

1. If you have not done so already, create the `blood` data set with the instructions given on page 300.
2. If you have not done so already, perform the one-way analysis of variance on page 302 and save the results in the object `anova.blood`.
3. Open the **Multiple Comparisons** dialog.
4. Select `anova.blood` as the **Model Object** from the pull-down menu.
5. We want to compare the levels of `diet` using Tukey's multiple comparison procedure. Select `diet` from the pull-down menu for **Levels Of** and set the **Method** to **Tukey**.
6. Click **OK** to generate the multiple comparisons.

The **Report** window displays the result:

```
95 % simultaneous confidence intervals for specified
linear combinations, by the Tukey method

critical point: 2.7987
response variable: time

intervals excluding 0 are flagged by '****'

      Estimate Std.Error Lower Bound Upper Bound
A-B -5.00e+000      1.53       -9.28      -0.725 ****
A-C -7.00e+000      1.53      -11.30      -2.720 ****
A-D -8.93e-014      1.45       -4.06       4.060
B-C -2.00e+000      1.37       -5.82       1.820
B-D  5.00e+000      1.28        1.42       8.580 ****
C-D  7.00e+000      1.28        3.42      10.600 ****
```

From the above results and from the plot of the confidence intervals, we can see that diets A and D produce significantly different blood coagulation times than diets C and B.

MIXED EFFECTS

Mixed effects models are regression or ANOVA models that include both fixed and random effects.

Linear

The **Linear Mixed Effects Models** dialog fits a linear mixed-effects model in the formulation of Laird and Ware (1982), but allows for nested random effects.

Fitting a linear mixed effects model

From the main menu, choose **Statistics ▶ Mixed Effects ▶ Linear**. The **Linear Mixed Effects Models** dialog opens, as shown in Figure 8.49.

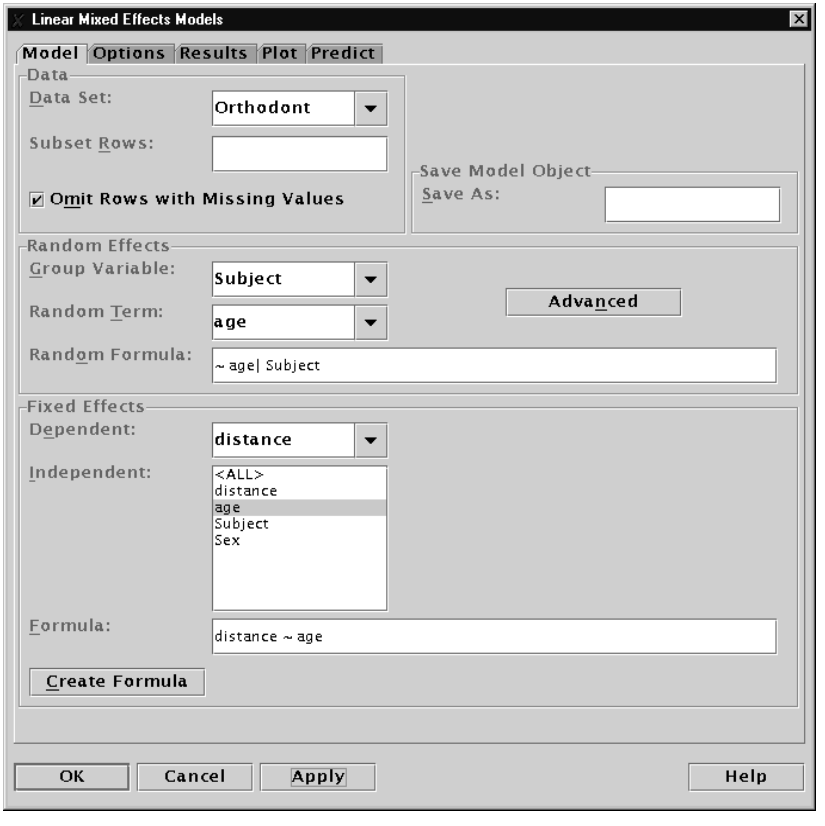


Figure 8.49: *The **Linear Mixed Effects Models** dialog.*

Example

The `Orthodont` data set has 108 rows and four columns, and contains an orthodontic measurement on eleven girls and sixteen boys at four different ages. We use a linear mixed-effects model to determine the change in distance with age. The model includes fixed and random effects of `age`, with `Subject` indicating the grouping of measurements.

1. Open the **Linear Mixed Effects Models** dialog.
2. Type `Orthodont` in the **Data Set** field.
3. Specify `distance~age` in the **Formula** field.
4. Select `Subject` as a **Group Variable** and `age` as a **Random Term**. The **Random Formula** field is automatically filled in as `~ age|Subject`.
5. Click **OK**.

A summary of the model is printed in the **Report** window.

Nonlinear

The **Nonlinear Mixed Effects Models** dialog fits a nonlinear mixed-effects model in the formulation described in Lindstrom and Bates (1990), but allows for nested random effects.

Fitting a nonlinear mixed effects model

From the main menu, choose **Statistics ► Mixed Effects ► Nonlinear**. The **Nonlinear Mixed Effects Models** dialog opens, as shown in Figure 8.50.

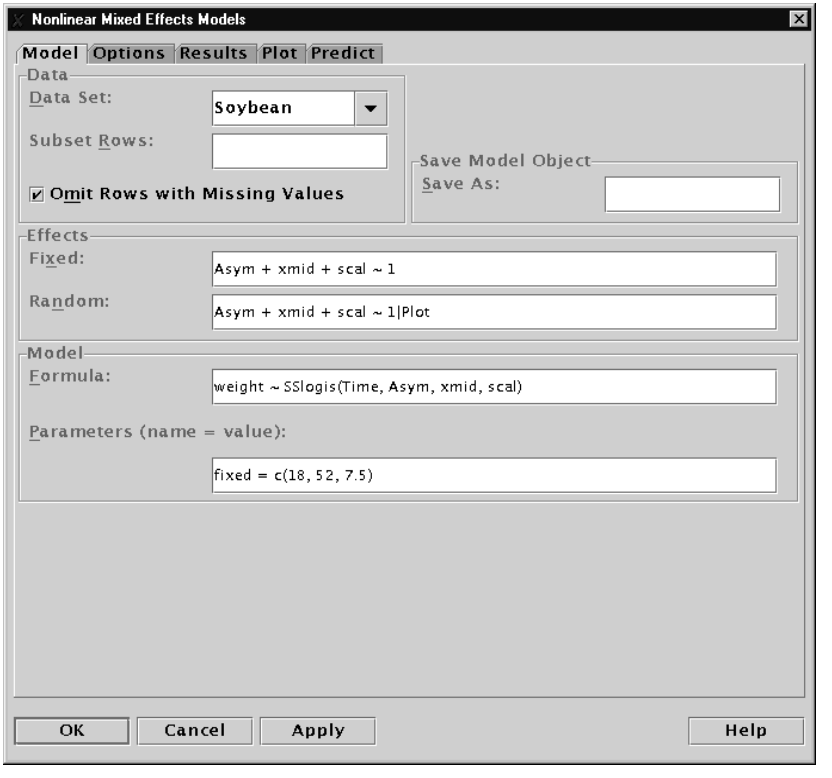


Figure 8.50: *The **Nonlinear Mixed Effects Models** dialog.*

Example

The `Soybean` data comes from an experiment that compares growth patterns of two genotypes of soybeans. Variables include a factor giving a unique identifier for each plot (`Plot`), a factor indicating which variety of soybean is in the plot (`Variety`), the year the plot was planted (`Year`), the time each sample was taken (`time`), and the average leaf weight per plant (`weight`). We are interested in modeling `weight` as a function of `Time` in a logistic model with parameters `Asym`, `xmid`, and `scal`. These parameters have both fixed and random effects. The grouping variable is `Plot`.

1. Open the **Nonlinear Mixed Effects Models** dialog.
2. Type `Soybean` in the **Data Set** field.
3. Type the following **Formula**:

   ```
   weight ~ SSlogis(Time, Asym, xmid, scal)
   ```

   This specifies that we want to predict `weight` by a function `SSlogis` of the variables `Time`, `Asym`, `xmid`, and `scal`. The `SSlogis` function is a *self-starting* function used to specify the nonlinear model, as well as provide initial estimates to the solver.
4. Specify starting fixed effect parameter estimates in the **Parameters** (**name=value**) field:

   ```
   fixed=c(18, 52, 7.5)
   ```
5. Specify that `Asym`, `xmid`, and `scal` are the fixed effects variables by typing the following formula in the **Fixed** field under **Effects**:

   ```
   Asym + xmid + scal ~ 1
   ```
6. Specify that `Asym`, `xmid`, and `scal` are the random effects variables and that `Plot` is the grouping variable by typing the following formula in the **Random** field under **Effects**:

   ```
   Asym + xmid + scal ~ 1 | Plot
   ```
7. Click **OK**.

A summary of the fitted model appears in the **Report** window.

GENERALIZED LEAST SQUARES

Generalized least squares models are regression or ANOVA models in which the residuals have a nonstandard covariance structure. The covariance structures supported include correlated and heteroscedastic residuals.

Linear

The **Generalized Least Squares** dialog fits a linear model using generalized least squares. Errors are allowed to be correlated and/or have unequal variances.

Performing generalized least squares regression

From the main menu, choose **Statistics ▶ Generalized Least Squares ▶ Linear**. The **Generalized Least Squares** dialog opens, as shown in Figure 8.51.

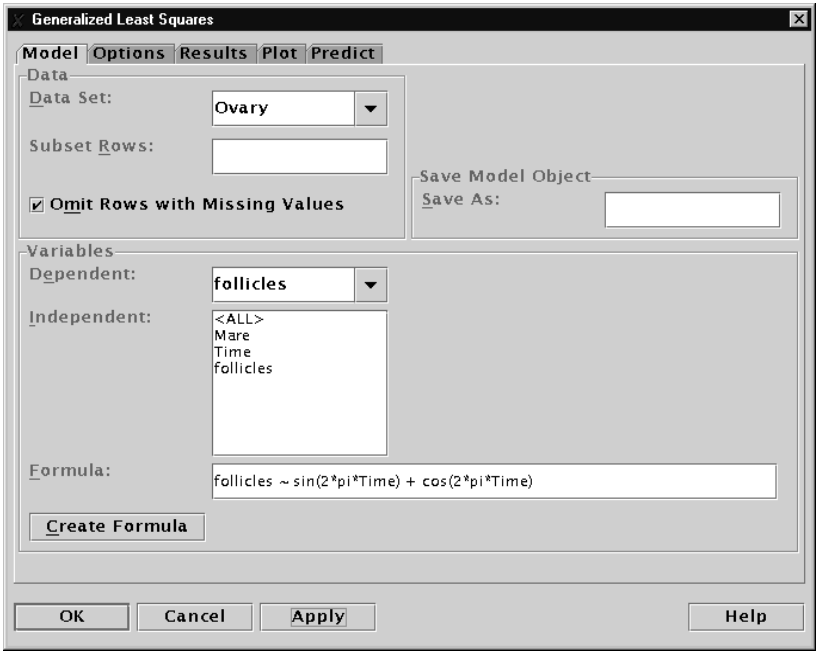


Figure 8.51: *The **Generalized Least Squares** dialog.*

Example

The `Ovary` data set has 308 rows and three columns giving the number of ovarian follicles detected in different mares at different times in their estrus cycles. Biological models suggest that the number of follicles may be modeled as a linear combination of the sine and cosine of `2*pi*Time`. We expect that the variation increases with `Time`, and hence use generalized least squares with a Power variance structure instead of standard linear regression. In a Power variance structure, the variance increases with a power of the absolute fitted values.

1. Open the **Generalized Least Squares** dialog.
2. Type `Ovary` in the **Data Set** field.
3. Enter the following **Formula**:

   ```
   follicles ~ sin(2*pi*Time) + cos(2*pi*Time)
   ```

4. On the **Options** page of the dialog, select **Power** as the **Variance Structure Type**.
5. Click **OK**.

A summary of the fitted model appears in the **Report** window.

Nonlinear

The **Generalized Nonlinear Least Squares** dialog fits a nonlinear model using generalized least squares. The errors are allowed to be correlated and/or have unequal variances.

Performing generalized nonlinear least squares regression

From the main menu, choose **Statistics ▶ Generalized Least Squares ▶ Nonlinear**. The **Generalized Nonlinear Least Squares** dialog opens, as shown in Figure 8.52.

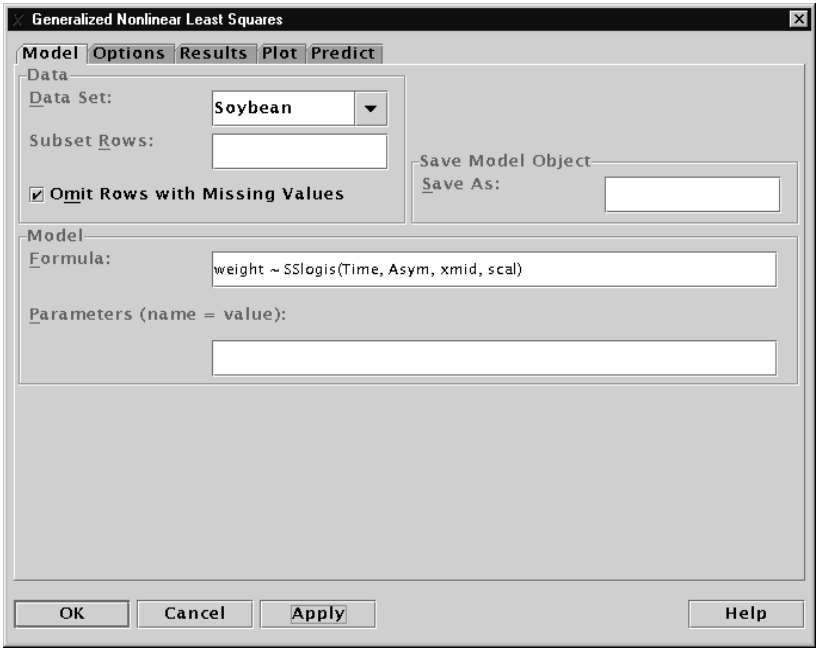


Figure 8.52: *The **Generalized Nonlinear Least Squares** dialog.*

Example

The `Soybean` data comes from an experiment to compare growth patterns of two genotypes of soybeans. Variables include a factor giving a unique identifier for each plot (`Plot`), a factor indicating which variety of soybean is in the plot (`Variety`), the year the plot was planted (`Year`), the time each sample was taken (`time`), and the average leaf weight per plant (`weight`). We are interested in modeling `weight` as a function of `Time` in a logistic model with parameters `Asym`, `xmid`, and `scal`. We expect that the variation increases with time, and hence use generalized least squares with a Power variance structure instead of standard nonlinear regression. In a Power variance structure, the variance increases with a power of the absolute fitted values.

1. Open the **Generalized Nonlinear Least Squares** dialog.
2. Type `Soybean` in the **Data Set** field.
3. Enter the following **Formula**:

   ```
   weight ~ SSlogis(Time, Asym, xmid, scal)
   ```

   The `SSlogis` function is a *self-starting* function used to specify the nonlinear model, as well as provide initial estimates to the solver.
4. On the **Options** page of the dialog, select **Power** as the **Variance Structure Type**.
5. Click **OK**.

A summary of the fitted model appears in the **Report** window.

SURVIVAL

Survival analysis is used for data in which censoring is present.

Nonparametric Survival

Nonparametric survival curves are estimates of the probability of survival over time. They are used in situations such as medical trials where the response is time to failure, usually with some times lost to censoring. The most commonly used nonparametric survival curve is the Kaplan-Meier estimate. The **Nonparametric Survival** dialog fits a variety of nonparametric survival curves and allows the inclusion of grouping variables.

Fitting a nonparametric survival curve

From the main menu, choose **Statistics** ▶ **Survival** ▶ **Nonparametric Survival**. The **Nonparametric Survival** dialog opens, as shown in Figure 8.53.

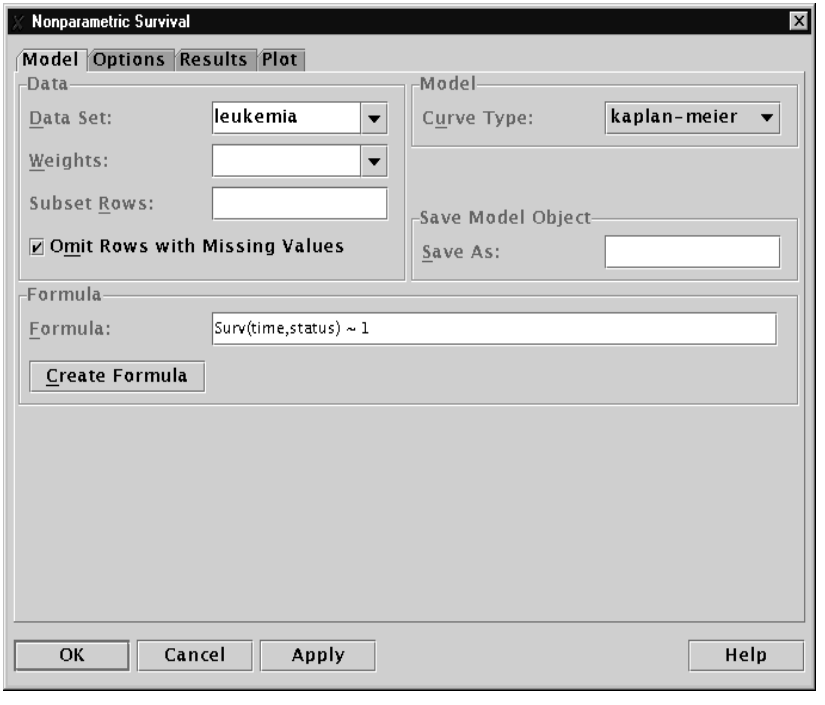


Figure 8.53: *The **Nonparametric Survival** dialog.*

Example

The `leukemia` data set contains data from a trial to evaluate efficacy of maintenance chemotherapy for acute myelogenous leukemia. We fit a Kaplan-Meier survival curve to the full set of data.

1. Open the **Nonparametric Survival** dialog.
2. Type `leukemia` in the **Data Set** field.
3. Enter the **Formula** `Surv(time,status)~1` or click on the **Create Formula** button to construct the formula. The `Surv` function creates a survival object, which is the appropriate response variable for a survival formula.
4. Click **OK**.

A summary of the fitted model appears in the **Report** window, and a plot of the survival curve with confidence intervals appears in a **Graph** window.

Cox Proportional Hazards

The *Cox proportional hazards model* is the most commonly used regression model for survival data. It allows the estimation of nonparametric survival curves (such as Kaplan-Meier curves) in the presence of covariates. The effect of the covariates upon survival is usually of primary interest.

Fitting a Cox proportional hazards model

From the main menu, choose **Statistics ► Survival ► Cox Proportional Hazards**. The **Cox Proportional Hazards** dialog opens, as shown in Figure 8.54.

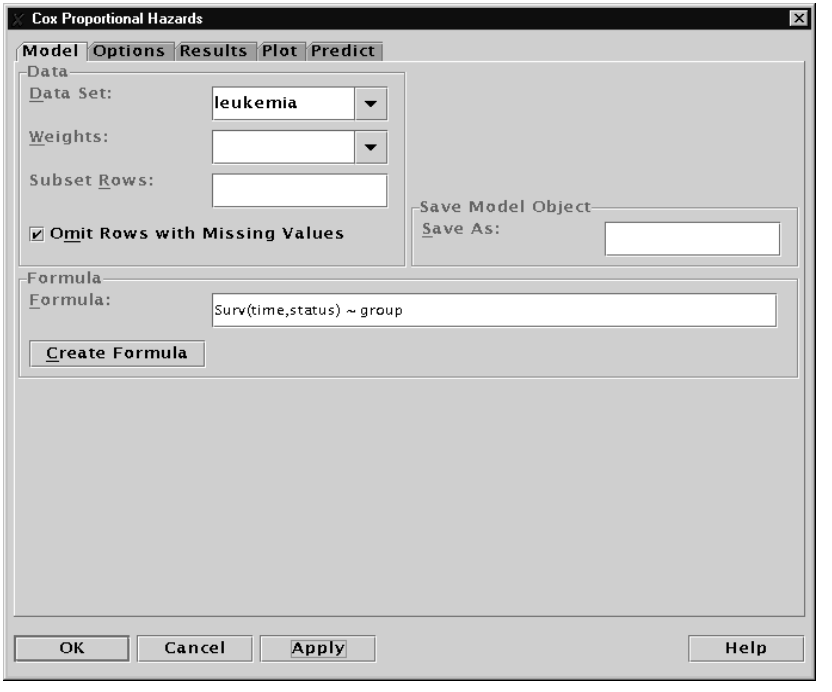


Figure 8.54: *The **Cox Proportional Hazards** dialog.*

Example

We fit a Cox proportional hazards model to the `leukemia` data set with `group` used as a covariate.

1. Open the **Cox Proportional Hazards** dialog.
2. Type `leukemia` in the **Data Set** field.
3. Enter the **Formula** `Surv(time,status)~group` or click the **Create Formula** button to construct the formula. The `Surv` function creates a survival object, which is the appropriate response variable for a survival formula.
4. Select the **Survival Curves** check box on the **Plot** page.
5. Click **OK**.

A summary of the fitted model appears in the **Report** window, and a plot of the survival curve with confidence intervals appears in a **Graph** window.

Parametric Survival

Parametric regression models for censored data are used in a variety of contexts ranging from manufacturing to studies of environmental contaminants. Because of their frequent use for modeling failure time or survival data, they are often referred to as *parametric survival models*. In this context, they are used throughout engineering to discover reasons why engineered products fail. They are called *accelerated failure time models* or *accelerated testing models* when the product is tested under more extreme conditions than normal to accelerate its failure time.

The **Parametric Survival** and **Life Testing** dialogs fit the same type of model. The difference between the two dialogs is in the options available. The **Life Testing** dialog supports threshold estimation, truncated distributions, and offsets. In addition, it provides a variety of diagnostic plots and the ability to obtain predicted values. This functionality is not available in the **Parametric Survival** dialog. In contrast, the **Parametric Survival** dialog supports frailty and penalized likelihood models, which is not available in the **Life Testing** dialog.

Fitting a parametric survival model

From the main menu, choose **Statistics ► Survival ► Parametric Survival**. The **Parametric Survival** dialog opens, as shown in Figure 8.55.

Figure 8.55: *The **Parametric Survival** dialog.*

Example

The `capacitor` data set contains measurements from a simulated accelerated life testing of capacitors. It includes time to failure (`days`), indicator of failure or censoring (`event`), and the voltage at which the test was run (`voltage`). We use a parametric survival model to examine how voltage influences the probability of failure.

1. Open the **Parametric Survival** dialog.
2. Type `capacitor` in the **Data Set** field.
3. Enter the **Formula** `Surv(days,event)~voltage` or click the **Create Formula** button to construct the formula. The `Surv` function creates a survival object, which is the appropriate response variable for a survival formula.
4. Click **OK**.

A summary of the fitted model appears in the **Report** window.

Life Testing

The **Life Testing** dialog fits a parametric regression model for censored data. These models are used in a variety of contexts ranging from manufacturing to studies of environmental contaminants. Because of their frequent use for modeling failure time or survival data, they are often referred to as *parametric survival models*. In this context, they are used throughout engineering to discover reasons why engineered products fail. They are called *accelerated failure time models* or *accelerated testing models* when the product is tested under more extreme conditions than normal to accelerate its failure time.

The **Parametric Survival** and **Life Testing** dialogs fit the same type of model. The difference between the two dialogs is in the options available. The **Life Testing** dialog supports threshold estimation, truncated distributions, and offsets. In addition, it provides a variety of diagnostic plots and the ability to obtain predicted values. This functionality is not available in the **Parametric Survival** dialog. In contrast, the **Parametric Survival** dialog supports frailty and penalized likelihood models, which is not available in the **Life Testing** dialog.

Performing life testing

From the main menu, choose **Statistics ▶ Survival ▶ Life Testing**. The **Life Testing** dialog opens, as shown in Figure 8.56.

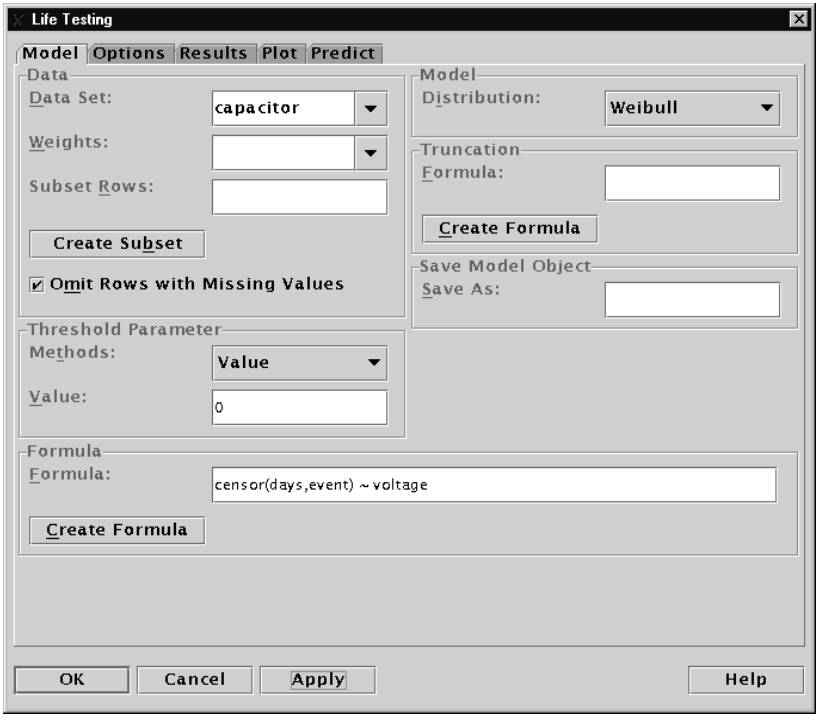


Figure 8.56: *The **Life Testing** dialog.*

Example

We use the **Life Testing** dialog to examine how voltage influences the probability of failure in the `capacitor` data set.

1. Open the **Life Testing** dialog.
2. Type `capacitor` in the **Data Set** field.
3. Enter the **Formula** `censor(days,event)~voltage` or click the **Create Formula** button to construct the formula. The `censor` function creates a survival object, which is the appropriate response variable for a survival formula. It is similar to the `Surv` function, but provides more options for specifying censor codes.
4. Click **OK**.

A summary of the fitted model appears in the **Report** window.

TREE

Tree-based models provide an alternative to linear and additive models for regression problems, and to linear and additive logistic models for classification problems. Tree models are fit by successively splitting the data to form homogeneous subsets. The result is a hierarchical tree of decision rules useful for prediction or classification.

Tree Models

The **Tree Models** dialog is used to fit a tree model.

Fitting a tree model

From the main menu, choose **Statistics ▶ Tree ▶ Tree Models**. The **Tree Models** dialog opens, as shown in Figure 8.57.

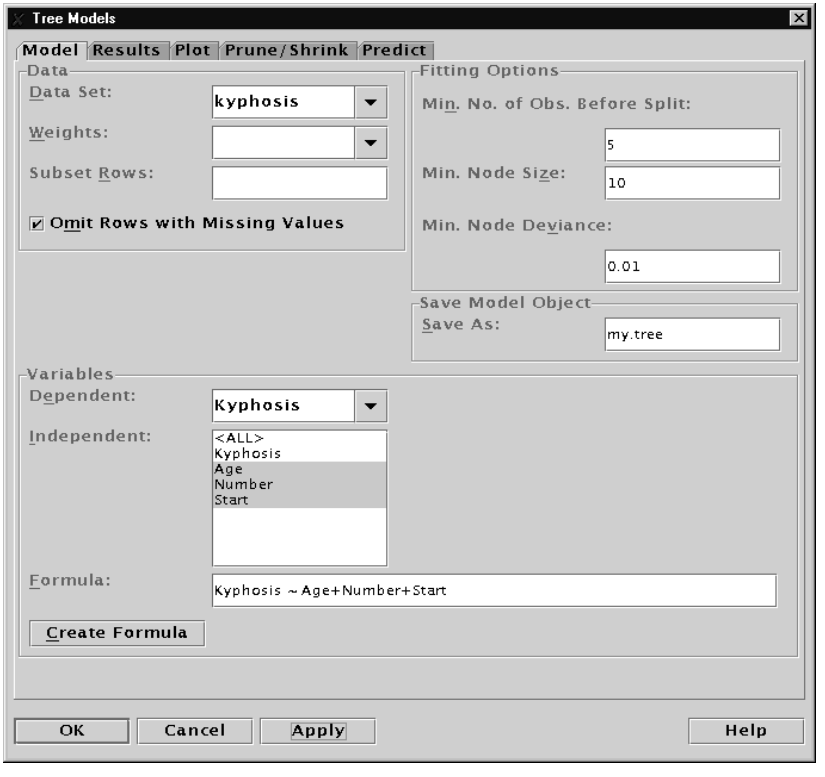


Figure 8.57: *The **Tree Models** dialog.*

Example

The kyphosis data set has 81 rows representing data on 81 children who have had corrective spinal surgery. The outcome Kyphosis is a binary variable, and the other three columns Age, Number, and Start, are numeric. Kyphosis is a post-operative deformity which is present in some children receiving spinal surgery. We are interested in examining whether the child's age, the number of vertebrae operated on, or the starting vertebra influence the likelihood of the child having a deformity.

We fit a classification tree to the data, in which a tree structure is used to classify individuals as likely or unlikely to have kyphosis based on their values of Age, Number, and Start. The resulting classification tree divides individuals into groups based on these variables.

1. Open the **Tree Models** dialog.
2. Type kyphosis in the **Data Set** field.
3. Specify Kyphosis~Age+Number+Start in the **Formula** field.
4. Type my.tree in the **Save As** field. A tree model object is saved under this name, which we explore in a later example using **Tree Tools**.
5. Click **OK**.

A summary of the model is printed in the **Report** window, and a tree plot is displayed in a **Graph** window.

Tree Tools

S-PLUS provides a rich suite of tools for interactively examining a regression tree. To use **Tree Tools**, first use the **Tree Models** dialog to create a tree model. Save the tree model by specifying a name in the **Save As** field of the dialog.

All of the **Tree Tools** begin by creating a plot of the specified tree model. The **Browse**, **Burl**, **Histogram**, **Identify**, and **Snip** tools let you select splits or nodes on the plot, and provide information on the selection. Click the left mouse button to make a selection, and click the right or center mouse button to leave the selection mode. With these tools, it may be necessary to arrange your windows prior to clicking **OK** or **Apply** so that the necessary **Graph** and **Report** windows are in view while making selections.

The tools behave in the following manner:

- **Browse**: select a node on the tree plot. Summary information on the node appears in the **Report** window. Right-click to leave the selection mode. Specify a name in the **Save As** field to save a list of the node information.
- **Burl**: select a split on the tree plot. Plots appear under the tree that display the change in deviance for all candidate splits. The actual split has the largest change in deviance. These plots are useful for examining whether other splits would produce an improvement in fit similar to the improvement from the actual split. Right-click to leave the selection mode. Specify a name in the **Save As** field to save a list with information on the candidate splits.
- **Histogram**: specify variables for which to draw histograms in the **Hist Variables** field. Select a split on the tree plot. Plots appear under the tree that display histograms of the specified variables, with separate histograms for the values in the two nodes resulting from the split. Right-click to leave the selection mode. Specify a name in the **Save As** field to save a list of the variable values corresponding to the histograms.
- **Identify**: select a node on the tree plot. The row names or numbers for the observations in that node appear in the **Report** window. Right-click to leave the selection mode. Specify a name in the **Save As** field to save a list of the observations in each node.
- **Rug**: specify the variable to plot in the **Rug/Tile Variable** field. A high-density plot that shows the average value of the specified variable for observations in each leaf is plotted beneath the tree plot. Specify a name in the **Save As** field to save a vector of the average values. This tool is not interactive.
- **Snip**: use this tool to create a new tree with some splits removed. Select a node on the tree plot to print the total tree deviance and what the total tree deviance would be if the subtree rooted at the node were removed. Click a second time on the same node to snip that subtree off and visually erase the subtree. This process may be repeated any number of times. Right-click to leave the selection mode. Specify a name in the **Save As** field to save the snipped tree.

- **Tile**: specify a variable to plot in the **Rug/Tile Variable** field. A vertical bar plot of the variable is plotted beneath the tree plot. Factor variables have one bar per level, and numeric variables are quantized into four equi-sized ordered levels. Specify a name in the **Save As** field to save a matrix of frequency counts for the observations in each leaf. This tool is not interactive.

Using the tree tools

From the main menu, choose **Statistics ▶ Tree ▶ Tree Tools**. The **Tree Tools** dialog opens, as shown in Figure 8.58.

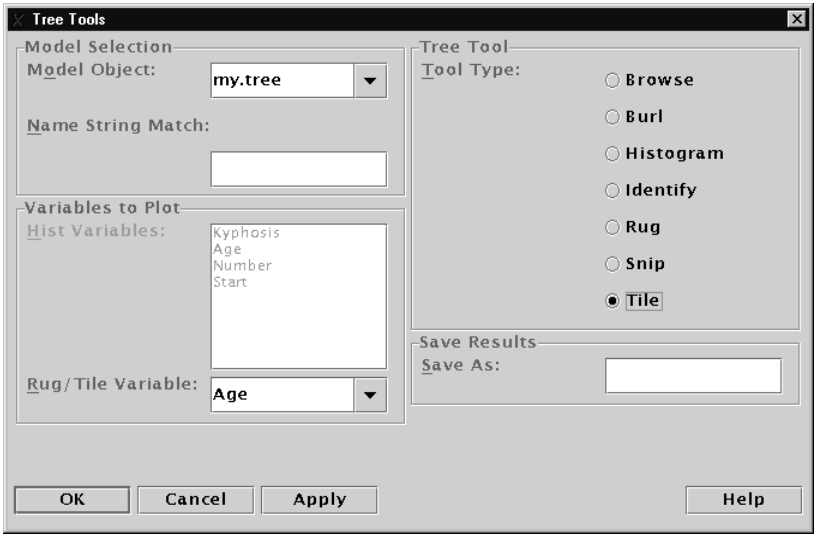


Figure 8.58: *The **Tree Tools** dialog.*

Example

In the section Tree Models on page 381, we fit a classification tree to the `kyphosis` data. We can use a tree tile plot to see histograms of `Age` within each group.

1. If you have not done so already, fit the classification tree and save the results in an object named `my.tree`. This process is outlined on page 382.
2. Open the **Tree Tools** dialog.

3. Select `my.tree` as the **Model Object**.
4. Select **Tile** as the **Tool Type**.
5. Select `Age` as the **Rug/Tile Variable**.
6. Click **OK**.

A tree tile plot is displayed in a **Graph** window. The top portion of the graph contains a plot of the tree. The bottom portion contains histograms of `Age` for each terminal node in the tree.

COMPARE MODELS

In regression and ANOVA, the data analyst often has a variety of candidate models of interest. From these models, the data analyst usually chooses one which is thought to best describe the relationship between the predictors and the response.

Model selection typically involves making a trade-off between complexity and goodness-of-fit. A more complex model (one involving more variables or interactions of variables) is guaranteed to fit the observed data more closely than a simpler model. For example, a model with as many parameters as observations would fit the data perfectly. However, as the model grows more complex, it begins to reflect the random variation in the sample obtained rather than a more general relationship between the response and the predictors. This may make the model less useful than a simpler one for predicting new values or drawing conclusions regarding model structure.

The general strategy in regression is to choose a simpler model when doing so does not reduce the goodness-of-fit by a significant amount. In linear regression and ANOVA, an F-test may be used to compare two models. In logistic and log-linear regression, a chi-square test comparing deviances is appropriate.

The **Compare Models** dialog lets you compare the goodness-of-fit of two or more models. Typically, the models should be *nested*, in that the simpler model is a special case of the more complex model. Before using the **Compare Models** dialog, first save the models of interest as objects.

Comparing models

From the main menu, choose **Statistics ► Compare Models**. The **Compare Models** (**Likelihood Ratio Test**) dialog opens, as shown in Figure 8.59.

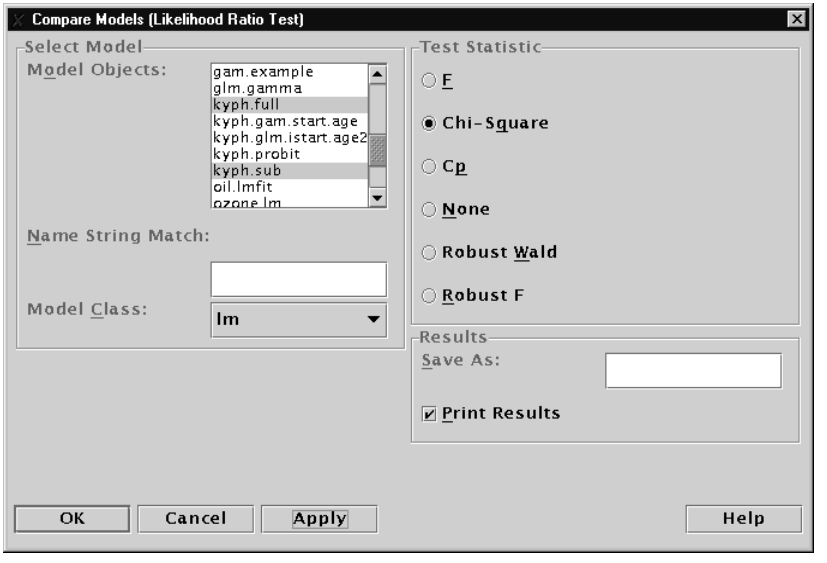


Figure 8.59: *The **Compare Models (Likelihood Ratio Test)** dialog.*

Example

In the kyphosis analysis of the section Logistic Regression, we suggested that `Start` had a significant effect upon `Kyphosis`, but `Age` and `Number` did not. We can use a chi-square test to determine whether a model with just `Start` is sufficient.

1. Open the **Logistic Regression** dialog.
2. Type `kyphosis` in the **Data Set** field.
3. Specify `Kyphosis~Age+Number+Start` in the **Formula** field. Type `kyph.full` in the **Save As** field and click **Apply**. Information describing this model is saved as an object named `kyph.full`.
4. Change the **Formula** field to `Kyphosis~Start`. Change the **Save As** name to `kyph.sub`, and click **OK**. Information describing this model is saved as an object named `kyph.sub`.
5. Open the **Compare Models** (**Likelihood Ratio Test**) dialog.

6. CTRL-click to select `kyph.full` and `kyph.sub` in the **Model Objects** list.
7. Select **Chi-Square** as the **Test Statistic**.
8. Click **OK**.

An analysis of deviance table appears in the **Report** window. The table displays the degrees of freedom and residual deviance for each model. Under the null hypothesis that the simpler model is appropriate, the difference in residual deviances is distributed as a chi-squared statistic. The `Pr(Chi)` column provides a p-value for the hypothesis that the simpler model is appropriate. If this value is less than a specific value, typically 0.05, then the more complex model causes a large enough change in deviance to warrant the inclusion of the additional terms. That is, the extra complexity is justified by an improvement in goodness-of-fit.

In our example the p-value of 0.035 suggests that `Age` and/or `Number` add extra information useful for predicting the outcome.

```
Analysis of Deviance Table

Response: Kyphosis

                 Terms Resid. Df Resid. Dev        Test
1 Age + Number + Start        77   61.37993
2                Start        79   68.07218 -Age-Number
  Df  Deviance    Pr(Chi)
1
2 -2 -6.692253 0.03522052
```

CLUSTER ANALYSIS

In cluster analysis, we search for groups (clusters) in the data in such a way that objects belonging to the same cluster resemble each other, whereas objects in different clusters are dissimilar.

Compute Dissimilarities

A data set for clustering can consist of either rows of observations, or a dissimilarity object storing measures of dissimilarities between observations. K-means, partitioning around medoids using the large data algorithm, and monothetic clustering all operate on a data set. Partitioning around medoids, fuzzy clustering, and the hierarchical methods take either a data set or a dissimilarity object.

The clustering routines themselves do not accept nonnumeric variables. If a data set contains nonnumeric variables such as factors, they must either be converted to numeric variables, or dissimilarities must be used.

How we compute the dissimilarity between two objects depends on the type of the original variables. By default, numeric columns are treated as interval-scaled variables, factors are treated as nominal variables, and ordered factors are treated as ordinal variables. Other variable types should be specified as such through the fields in the **Special Variable Types** group.

Calculating dissimilarities

From the main menu, choose **Statistics ► Cluster Analysis ► Compute Dissimilarities**. The **Compute Dissimilarities** dialog opens, as shown in Figure 8.60.

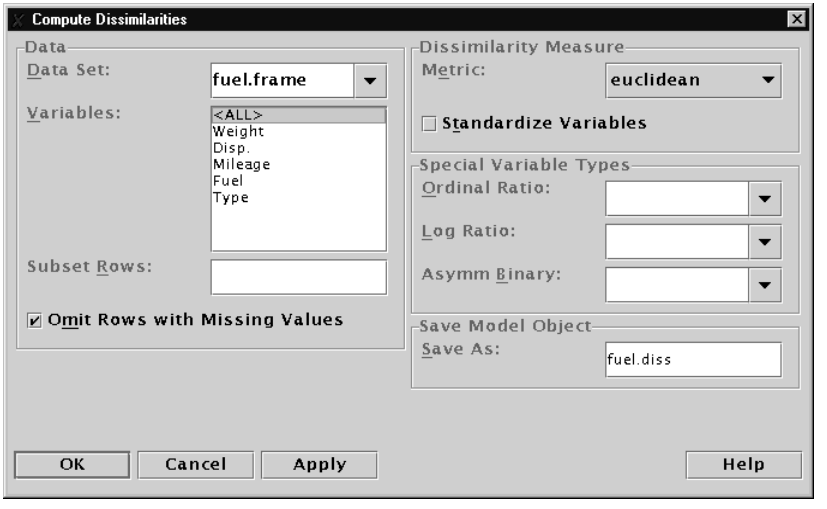


Figure 8.60: *The **Compute Dissimilarities** dialog.*

Example

The data set `fuel.frame` is taken from the April 1990 issue of *Consumer Reports*. It contains 60 observations (rows) and 5 variables (columns). Observations of weight, engine displacement, mileage, type, and fuel were taken for each of sixty cars. In the `fuel.frame` data, we calculate dissimilarities as follows:

1. Open the **Compute Dissimilarities** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. Type `fuel.diss` in the **Save As** field.
4. Click **OK**.

The dissimilarities are calculated and saved in the object `fuel.diss`. We use this object in later examples of clustering dialogs.

K-Means Clustering

One of the most well-known partitioning methods is *k-means*. In the k-means algorithm, observations are classified as belonging to one of k groups. Group membership is determined by calculating the centroid for each group (the multidimensional version of the mean) and assigning each observation to the group with the closest centroid.

Performing k-means clustering

From the main menu, choose **Statistics ▶ Cluster Analysis ▶ K-Means**. The **K-Means Clustering** dialog opens, as shown in Figure 8.61.

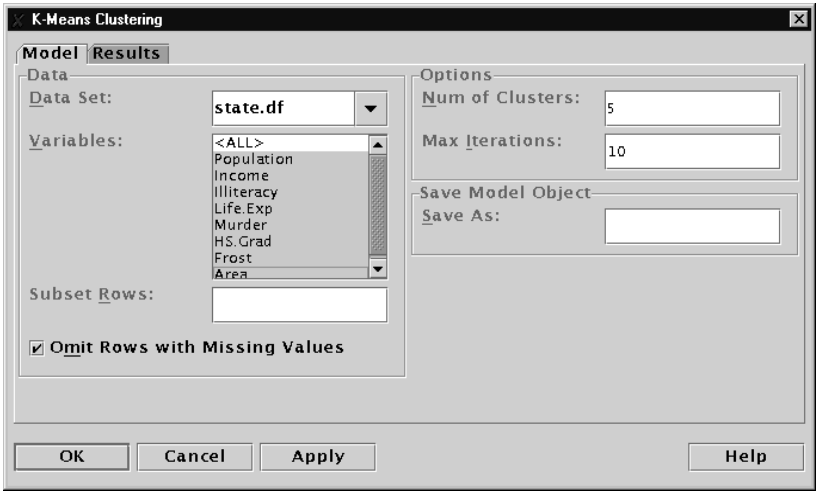


Figure 8.61: *The **K-Means Clustering** dialog.*

Example

We cluster the information in the `state.x77` data set. These data describe various characteristics of the 50 states, including population, income, illiteracy, life expectancy, and education. By default, `state.x77` is stored in an object of class `"matrix"`. We must therefore convert it to class `"data.frame"` before it can be recognized by the dialogs. To do this, type the following in the **Commands** window:

```
> state.df <- data.frame(state.x77)
```

We can now proceed with the k-means clustering analysis on the `state.df` data frame:

1. Open the **K-Means Clustering** dialog.
2. Type `state.df` in the **Data Set** field.

3. CTRL-click to select the **Variables** `Population` through `Area`.
4. Click **OK**.

A summary of the clustering appears in the **Report** window.

Partitioning Around Medoids

The *partitioning around medoids* algorithm is similar to k-means, but it uses medoids rather than centroids. Partitioning around medoids has the following advantages: it accepts a dissimilarity matrix; it is more robust because it minimizes a sum of dissimilarities instead of a sum of squared Euclidean distances; and it provides novel graphical displays (silhouette plots and clusplots).

Performing partitioning around medoids

From the main menu, choose **Statistics ▶ Cluster Analysis ▶ Partitioning Around Medoids**. The **Partitioning Around Medoids** dialog opens, as shown in Figure 8.62.

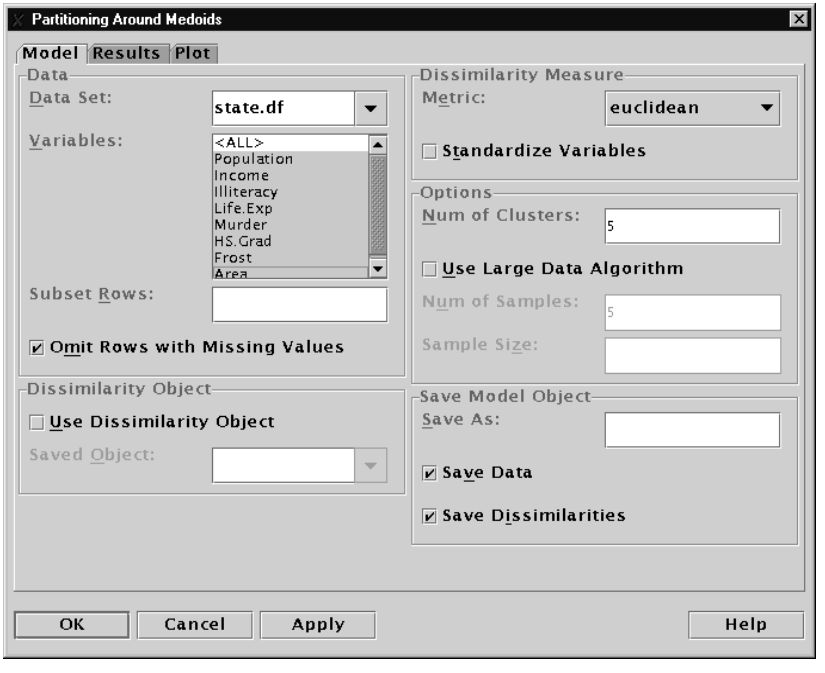


Figure 8.62: *The **Partitioning Around Medoids** dialog.*

Example 1

In the section K-Means Clustering on page 390, we clustered the information in the `state.df` data set using the k-means algorithm. In this example, we use the partitioning around medoids algorithm.

1. If you have not already done so, create the `state.df` data frame from the `state.x77` matrix. The instructions for doing this are located on page 391.
2. Open the **Partitioning Around Medoids** dialog.
3. Type `state.df` in the **Data Set** field.
4. CTRL-click to select the **Variables** `Population` through `Area`.
5. Click **OK**.

A summary of the clustering appears in the **Report** window.

Example 2

In the section Compute Dissimilarities on page 389, we calculated dissimilarities for the `fuel.frame` data set. In this example, we cluster the `fuel.frame` dissimilarities using the partitioning around medoids algorithm.

1. If you have not already done so, create the object `fuel.diss` from the instructions on page 390.
2. Open the **Partitioning Around Medoids** dialog.
3. Select the **Use Dissimilarity Object** check box.
4. Select `fuel.diss` as the **Saved Object**.
5. Click **OK**.

A summary of the clustering appears in the **Report** window.

Fuzzy Partitioning

Most clustering algorithms are crisp clustering methods. This means that each object of the data set is assigned to exactly one cluster. For instance, an object lying between two clusters must be assigned to one of them. In *fuzzy clustering*, each observation is given fractional membership in multiple clusters.

Performing fuzzy partitioning

From the main menu, choose **Statistics ▶ Cluster Analysis ▶ Fuzzy Partitioning**. The **Fuzzy Partitioning** dialog opens, as shown in Figure 8.63.

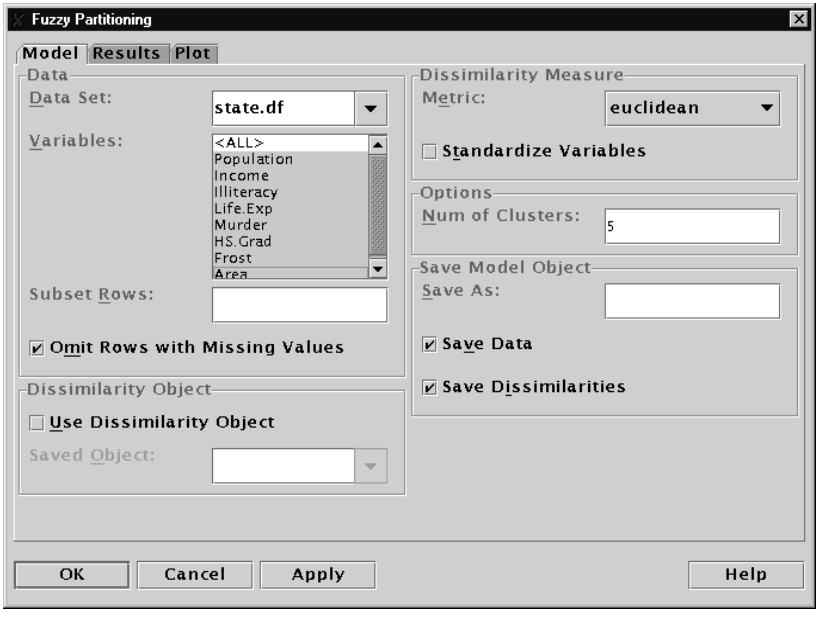


Figure 8.63: *The **Fuzzy Partitioning** dialog.*

Example 1

In the section K-Means Clustering on page 390, we clustered the information in the `state.df` data set using the k-means algorithm. In this example, we use fuzzy partitioning.

1. If you have not already done so, create the `state.df` data frame from the `state.x77` matrix. The instructions for doing this are located on page 391.
2. Open the **Fuzzy Partitioning** dialog.
3. Type `state.df` in the **Data Set** field.
4. CTRL-click to select the **Variables** `Population` through `Area`, and click **OK**.

A summary of the clustering appears in the **Report** window.

Example 2

In the section Compute Dissimilarities on page 389, we calculated dissimilarities for the `fuel.frame` data set. In this example, we cluster the `fuel.frame` dissimilarities using fuzzy partitioning.

1. If you have not already done so, create the object `fuel.diss` from the instructions on page 390.
2. Open the **Fuzzy Partitioning** dialog.
3. Select the **Use Dissimilarity Object** check box.
4. Select `fuel.diss` as the **Saved Object**.
5. Click **OK**.

A summary of the clustering appears in the **Report** window.

Agglomerative Hierarchical Clustering

Hierarchical algorithms proceed by combining or dividing existing groups, producing a hierarchical structure that displays the order in which groups are merged or divided. *Agglomerative* methods start with each observation in a separate group, and proceed until all observations are in a single group.

Performing agglomerative hierarchical clustering

From the main menu, choose **Statistics ▶ Cluster Analysis ▶ Agglomerative Hierarchical**. The **Agglomerative Hierarchical Clustering** dialog opens, as shown in Figure 8.64.

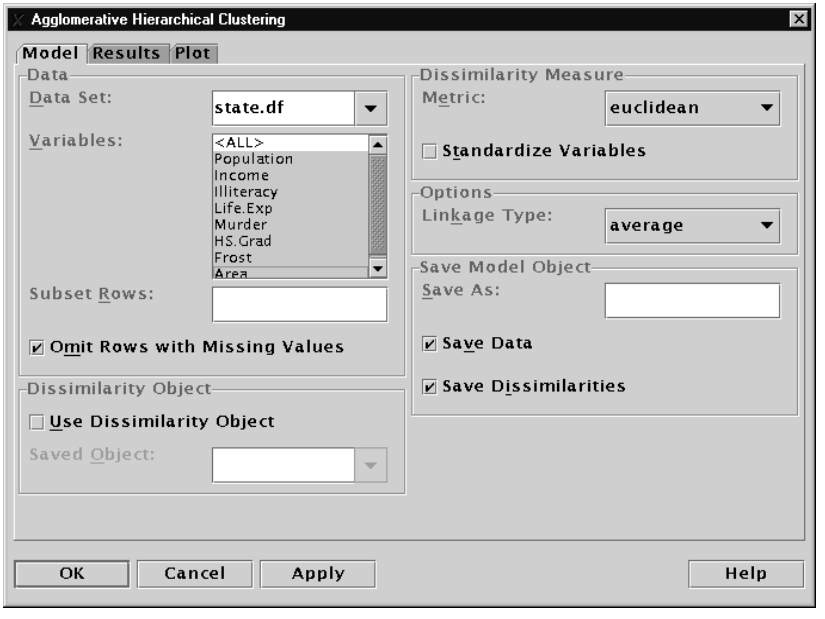


Figure 8.64: *The **Agglomerative Hierarchical Clustering** dialog.*

Example 1

In the section K-Means Clustering on page 390, we clustered the information in the `state.df` data set using the k-means algorithm. In this example, we use an agglomerative hierarchical method.

1. If you have not already done so, create the `state.df` data frame from the `state.x77` matrix. The instructions for doing this are located on page 391.
2. Open the **Agglomerative Hierarchical Clustering** dialog.
3. Type `state.df` in the **Data Set** field.
4. CTRL-click to select the **Variables** `Population` through `Area`.
5. Click **OK**.

A summary of the clustering appears in the **Report** window.

Example 2

In the section Compute Dissimilarities on page 389, we calculated dissimilarities for the `fuel.frame` data set. In this example, we cluster the `fuel.frame` dissimilarities using the agglomerative hierarchical algorithm.

1. If you have not already done so, create the object `fuel.diss` from the instructions on page 390.
2. Open the **Agglomerative Hierarchical Clustering** dialog.
3. Select the **Use Dissimilarity Object** check box.
4. Select `fuel.diss` as the **Saved Object**.
5. Click **OK**.

A summary of the clustering appears in the **Report** window.

Divisive Hierarchical Clustering

Hierarchical algorithms proceed by combining or dividing existing groups, producing a hierarchical structure that displays the order in which groups are merged or divided. *Divisive* methods start with all observations in a single group and proceed until each observation is in a separate group.

Performing divisive hierarchical clustering

From the main menu, choose **Statistics ▶ Cluster Analysis ▶ Divisive Hierarchical**. The **Divisive Hierarchical Clustering** dialog opens, as shown in Figure 8.65.

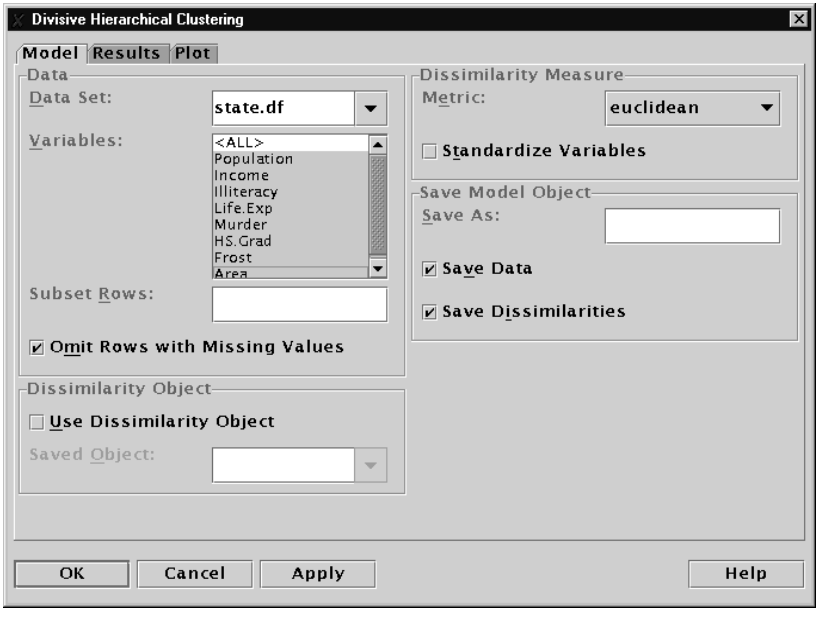


Figure 8.65: *The **Divisive Hierarchical Clustering** dialog.*

Example 1

In the section K-Means Clustering on page 390, we clustered the information in the `state.df` data set using the k-means algorithm. In this example, we use a divisive hierarchical method.

1. If you have not already done so, create the `state.df` data frame from the `state.x77` matrix. The instructions for doing this are located on page 391.
2. Open the **Divisive Hierarchical Clustering** dialog.
3. Type `state.df` in the **Data Set** field.
4. CTRL-click to select the **Variables** `Population` through `Area`.
5. Click **OK**.

A summary of the clustering appears in the **Report** window.

Example 2

In the section Compute Dissimilarities on page 389, we calculated dissimilarities for the `fuel.frame` data set. In this example, we cluster the `fuel.frame` dissimilarities using the divisive hierarchical algorithm.

1. If you have not already done so, create the object `fuel.diss` from the instructions on page 390.
2. Open the **Divisive Hierarchical Clustering** dialog.
3. Select the **Use Dissimilarity Object** check box.
4. Select `fuel.diss` as the **Saved Object**.
5. Click **OK**.

A summary of the clustering appears in the **Report** window.

Monothetic Clustering

When all of the variables in a data set are binary, a natural way to divide the observations is by splitting the data into two groups based on the two values of a particular binary variable. *Monothetic analysis* produces a hierarchy of clusters in which a group is split in two at each step, based on the value of one of the binary variables.

Performing monothetic clustering

From the main menu, choose **Statistics ▶ Cluster Analysis ▶ Monothetic** (**Binary Variables**). The **Monothetic Clustering** dialog opens, as shown in Figure 8.66.

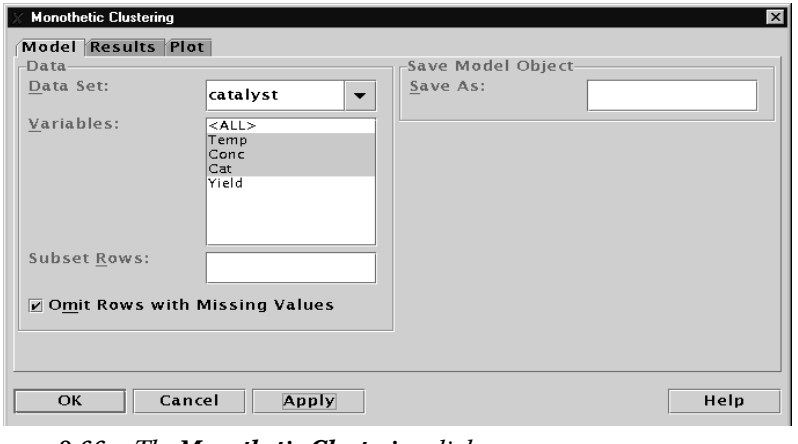


Figure 8.66: *The **Monothetic Clustering** dialog.*

Example

The catalyst data set comes from a designed experiment. Its eight rows represent all possible combinations of two temperatures (Temp), two concentrations (Conc), and two catalysts (Cat). The fourth column represents the response variable Yield. We are interested in determining how temperature, concentration, and catalyst affect the Yield. Before fitting a model to these data, we can group observations according to the three binary predictors by using monothetic clustering.

1. Open the **Monothetic Clustering** dialog.
2. Type catalyst in the Data Set field.
3. CTRL-click to highlight the **Variables** Temp, Conc, and Cat.
4. Click OK.

A summary of the monothetic clustering appears in the **Report** window.

MULTIVARIATE

Multivariate techniques summarize the structure of multivariate data based on certain classical models.

Discriminant Analysis

The **Discriminant Analysis** dialog lets you fit a linear or quadratic discriminant function to a set of feature data.

Performing discriminant analysis

From the main menu, choose **Statistics ► Multivariate ► Discriminant Analysis**. The **Discriminant Analysis** dialog opens, as shown in Figure 8.67.

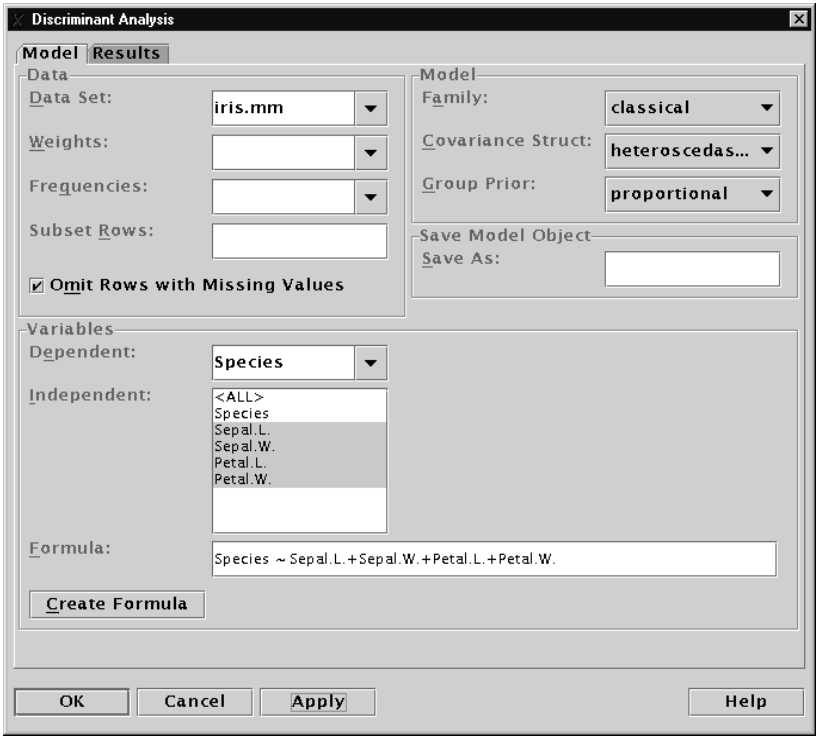


Figure 8.67: *The **Discriminant Analysis** dialog.*

Example

We perform a discriminant analysis on Fisher's `iris` data. This data set is a three-dimensional array giving 4 measurements on 50 flowers from each of 3 species of iris. The measurements are in centimeters and include sepal length, sepal width, petal length, and petal width. The iris species are Setosa, Versicolor, and Virginica.

Before performing the discriminant analysis, we must create a two-dimensional data frame that can be accepted by the dialogs. To do this, type the following in the **Commands** window:

```
> iris.mm <- data.frame(Species=factor(c(rep(1,50),
+ rep(2,50), rep(3,50)), labels=dimnames(iris)[[3]]),
+ rbind(iris[,,1], iris[,,2], iris[,,3])*10)
```

We can now use the **Discriminant Analysis** dialog on the `iris.mm` data frame:

1. Open the **Discriminant Analysis** dialog.
2. Type `iris.mm` in the **Data Set** field.
3. Choose `Species` as the **Dependent** variable.
4. CTRL-click to select `Sepal.L.`, `Sepal.W.`, `Petal.L.`, and `Petal.W.` as the **Independent** variables.
5. Choose **heteroscedastic** as the **Covariance Struct**.
6. Click **OK**.

A summary of the fitted model appears in the **Report** window.

Factor Analysis

In many scientific fields, notably psychology and other social sciences, you are often interested in quantities like intelligence or social status, which are not directly measurable. However, it is often possible to measure other quantities that reflect the underlying variable of interest. *Factor analysis* is an attempt to explain the correlations between observable variables in terms of underlying factors, which are themselves not directly observable. For example, measurable quantities, such as performance on a series of tests, can be explained in terms of an underlying factor, such as intelligence.

Performing factor analysis

From the main menu, choose **Statistics ▶ Multivariate ▶ Factor Analysis**. The **Factor Analysis** dialog opens, as shown in Figure 8.68.

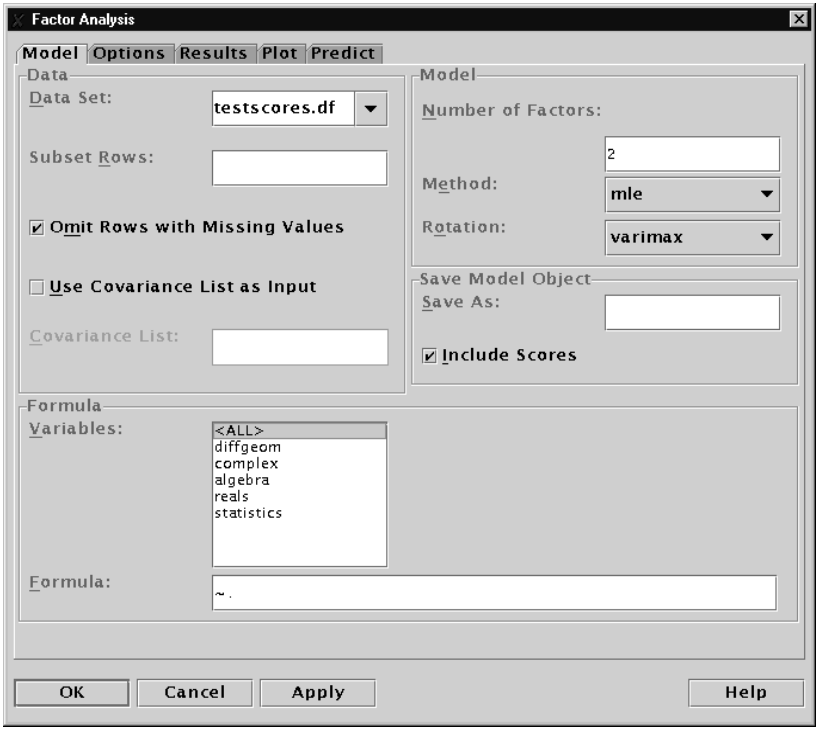


Figure 8.68: *The **Factor Analysis** dialog.*

Example

The data set `testscores` contains five test scores for each of twenty-five students. We use factor analysis to look for structure in the scores. By default, `testscores` is stored in an object of class `"matrix"`. We must therefore convert it to class `"data.frame"` before it can be recognized by the dialogs. To do this, type the following in the **Commands** window:

```
> testscores.df <- data.frame(testscores)
```

We can now proceed with the factor analysis on the `testscores.df` data frame:

1. Open the **Factor Analysis** dialog.
2. Type `testscores.df` in the **Data Set** field.
3. Specify that we want **2** factors in the **Number of Factors** field.
4. Select **<ALL>** in the **Variables** field.
5. Click **OK**.

A summary of the factor analysis appears in the **Report** window.

Principal Components

For investigations involving a large number of observed variables, it is often useful to simplify the analysis by considering a smaller number of linear combinations of the original variables. For example, scholastic achievement tests typically consist of a number of examinations in different subject areas. In attempting to rate students applying for admission, college administrators frequently reduce the scores from all subject areas to a single, overall score. *Principal components* is a standard technique for finding optimal linear combinations of the variables.

Performing principal components

From the main menu, choose **Statistics ▶ Multivariate ▶ Principal Components**. The **Principal Components** dialog opens, as shown in Figure 8.69.

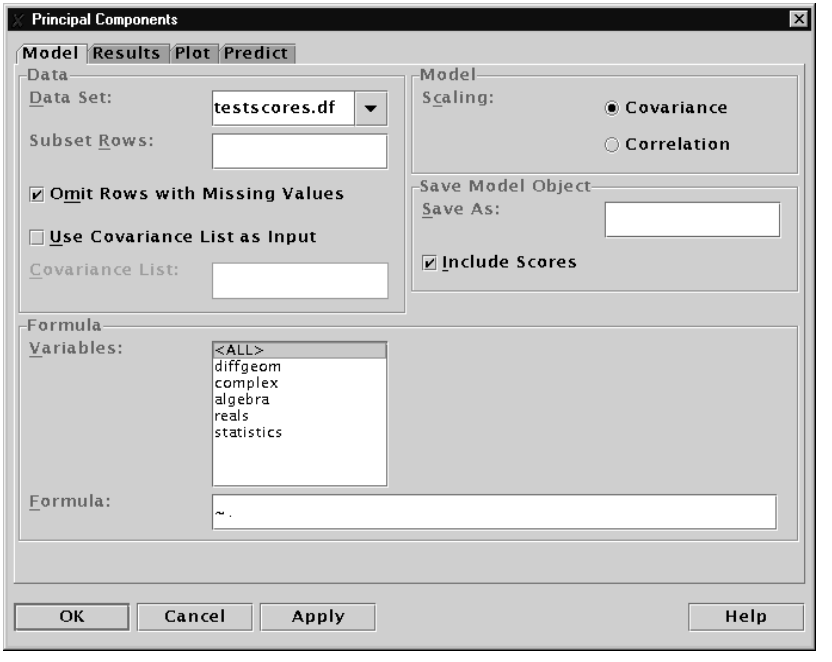


Figure 8.69: *The **Principal Components** dialog.*

Example

In the section Factor Analysis on page 402, we performed a factor analysis for the `testscores.df` data set. In this example, we perform a principal components analysis for these data.

1. If you have not done so already, create the `testscores.df` data frame with the instructions given on page 403.

1. Open the **Principal Components** dialog.
2. Type `testscores.df` in the **Data Set** field.
3. Select **<ALL>** in the **Variables** field.
4. Click **OK**.

A summary of the principal components analysis appears in the **Report** window.

MANOVA

Multivariate analysis of variance, known as MANOVA, is the extension of analysis of variance techniques to multiple responses. The responses for an observation are considered as one multivariate observation, rather than as a collection of univariate responses. If the responses are independent, then it is sensible to just perform univariate analyses. However, if the responses are correlated, then MANOVA can be more informative than the univariate analyses, as well as less repetitive.

Performing MANOVA

From the main menu, choose **Statistics ► Multivariate ► MANOVA**. The **Multivariate Analysis of Variance** dialog opens, as shown in Figure 8.70.

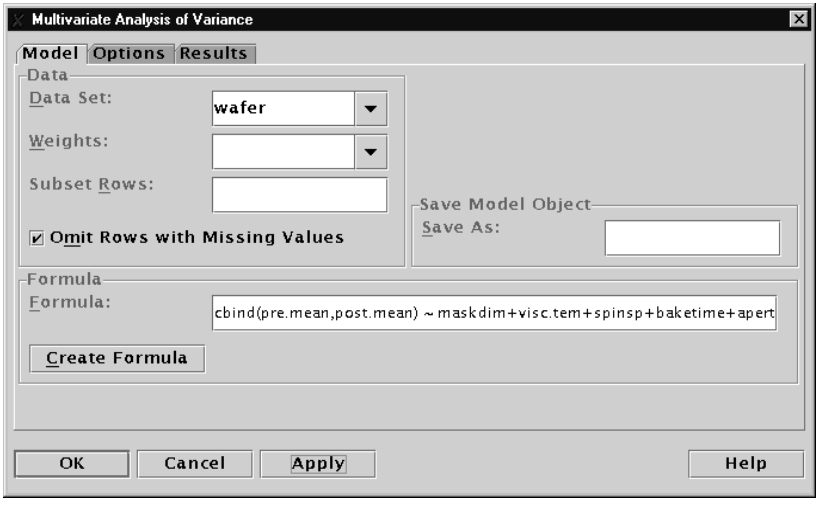


Figure 8.70: *The **Multivariate Analysis of Variance** dialog.*

Example

The data set `wafer` has eighteen rows and thirteen columns, of which eight contain factors, four contain responses, and one is the auxiliary variable N. It is a design object based on an orthogonal-array design for an experiment in which two integrated circuit wafers were made for each combination of factors. On each wafer, the pre- and post-etch line widths were measured five times. The response variables are the

mean and deviance of the measurements. As three of the wafers were broken, the auxiliary variable *N* gives the number of measurements actually made.

We are interested in treating the `pre.mean` and `post.mean` variables as a multivariate response, using MANOVA to explore the effect of each factor upon the response.

1. Open the **Multivariate Analysis of Variance** dialog.
2. Type `wafer` in the **Data Set** field.
3. Click the **Create Formula** button to open the **Formula** builder.
4. While holding down the CTRL key, select `pre.mean` and `post.mean` in the **Variables** list. Click the **Response** button to add these variables to the **Formula** as the response.
5. Select `maskdim`. Scroll through the **Variables** list until `etchtime` appears. Hold down **Shift** and select `etchtime`. This selects all columns between `maskdim` and `etchtime`. Click the **Main Effect** button to add these variables to the **Formula** as predictors.
6. Click **OK** to dismiss the **Formula** builder. The **Formula** field of the **MANOVA** dialog contains the formula you constructed.
7. Click **OK**.

A summary of the MANOVA appears in the **Report** window.

QUALITY CONTROL CHARTS

Quality control charts are useful for monitoring process data. *Continuous grouped* quality control charts monitor whether a process is staying within control limits. *Continuous ungrouped* charts are appropriate when variation is determined using sequential variation rather than group variation. It is also possible to create quality control charts for *counts* (the number of defective samples) and *proportions* (proportion of defective samples).

Continuous Grouped

The **Quality Control Charts** (**Continuous Grouped**) dialog creates quality control charts of means (xbar), standard deviations (s), and ranges (r).

Creating quality control charts (continuous grouped)

From the main menu, choose **Statistics** ▶ **Quality Control Charts** ▶ **Continuous Grouped**. The **Quality Control Charts** (**Continuous Grouped**) dialog opens, as shown in Figure 8.71.

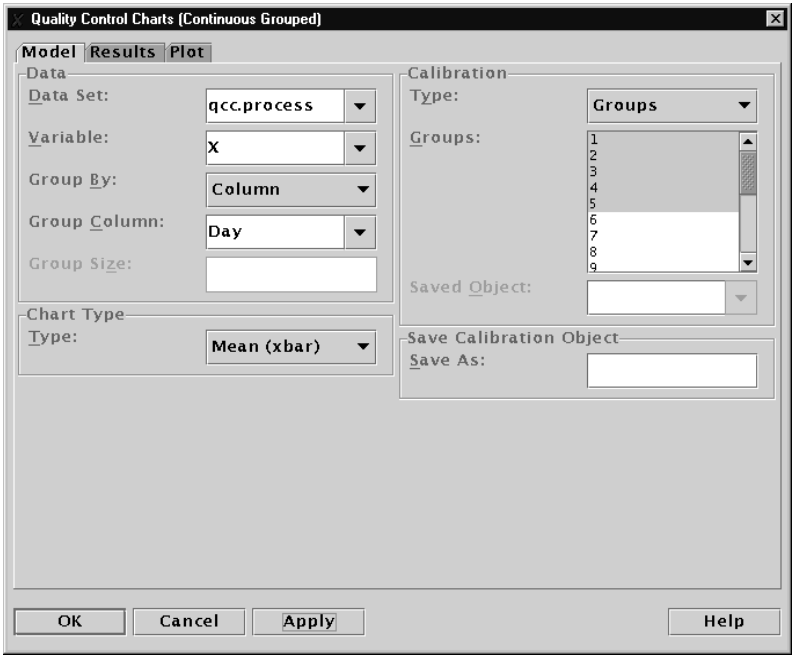


Figure 8.71: *The **Quality Control Charts (Continuous Grouped)** dialog.*

Example

In the section Kolmogorov-Smirnov Goodness-of-Fit on page 283, we created a data set called `qcc.process` that contains a simulated process with 200 measurements. Ten measurements per day were taken for a total of twenty days. In this example, we create an xbar Shewhart chart to monitor whether the process is staying within control limits. The first five days of observations are treated as calibration data for use in setting the control limits.

1. If you have not done so already, create the `qcc.process` data set with the instructions given on page 284.
2. Open the **Quality Control Charts** (**Continuous Grouped**) dialog.
3. Type `qcc.process` in the **Data Set** field.
4. Select `X` as the **Variable**.
5. Select `Day` as the **Group Column**.
6. Select **Groups** as the **Calibration Type**.
7. CTRL-click to select **1**, **2**, **3**, **4**, **5** from the **Groups** list box.
8. Click **OK**.

A Shewhart chart of the `X` data grouped by `Day` appears in a **Graph** window.

Continuous Ungrouped

The **Quality Control Charts** (**Continuous Ungrouped**) dialog creates quality control charts of exponentially weighted moving averages (ewma), moving averages (ma), moving standard deviations (ms), and moving ranges (mr). These charts are appropriate when variation is determined using sequential variation rather than group variation.

Creating quality control charts (continuous ungrouped)

From the main menu, choose **Statistics ▶ Quality Control Charts ▶ Continuous Ungrouped**. The **Quality Control Charts** (**Continuous Ungrouped**) dialog opens, as shown in Figure 8.72.

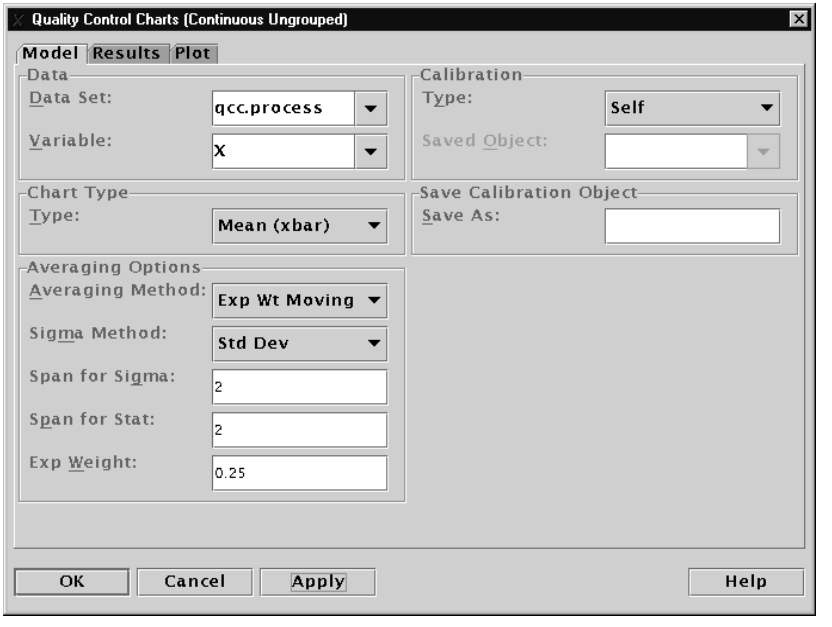


Figure 8.72: *The **Quality Control Charts (Continuous Ungrouped)** dialog.*

Example

For this example, we ignore the fact that `qcc.process` contains grouped data, and instead pretend that the 200 observations are taken at sequential time points. We create an exponentially weighted moving average Shewhart chart to monitor whether the process is staying within control limits.

1. If you have not done so already, create the `qcc.process` data set with the instructions given on page 284.
2. Open the **Quality Control Charts** (**Continuous Ungrouped**) dialog.
3. Type `qcc.process` in the **Data Set** field.
4. Select `X` as the **Variable**.
5. Click **OK**.

A Shewhart chart appears in a **Graph** window.

Counts and Proportions

The **Quality Control Charts** (**Counts and Proportions**) dialog creates quality control charts for counts (number of defective samples) and proportions (proportion of defective samples).

Creating quality control charts (counts and proportions)

From the main menu, choose **Statistics ▶ Quality Control Charts ▶ Counts and Proportions**. The **Quality Control Charts** (**Counts and Proportions**) dialog opens, as shown in Figure 8.73.

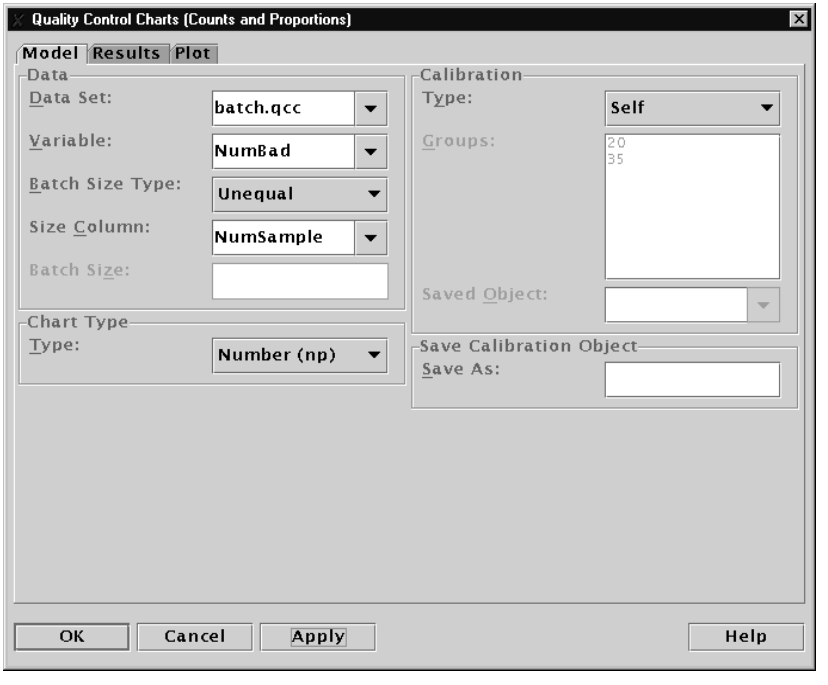


Figure 8.73: *The **Quality Control Charts (Counts and Proportions)** dialog.*

Example

We create an S-PLUS data set, `batch.qcc`, that contains simulated data representing the number of defective items in daily batches over 40 days. For the first 10 days the batches were of size 20, but for the remaining 30 days batches of 35 were taken. To create `batch.qcc`, type the following in the **Commands** window:

```
> NumSample <- c(rep(20,times=10), rep(35,times=30))
```

```
> NumBad <- scan()

1: 3 2 7 4 5 4 4 8
9: 3 4 6 6 6 9 18 9
17: 7 11 11 9 10 10 14 5
25: 15 11 14 15 11 10 14 8
33: 11 13 16 14 19 13 15 23
41:

> batch.qcc <- data.frame(NumBad,NumSample)
> batch.qcc

   NumBad NumSample
 1      3        20
 2      2        20
 3      7        20
 4      4        20
 5      5        20
 6      4        20
 7      4        20
 8      8        20
 9      3        20
10      4        20
11      6        35
12      6        35
13 . . .
```

The `NumBad` column encodes the number of defective items, and the `NumSample` column encodes the size of the batches.

We create a Number (np) Shewhart chart for these data.

1. Open the **Quality Control Charts (Counts and Proportions)** dialog.
2. Type `batch.qcc` in the **Data Set** field.
3. Select `NumBad` as the **Variable**.
4. Select `NumSample` as the **Size Column**.
5. Select **Number (np)** as the **Chart Type**.
6. Click **OK**.

A Shewhart chart of the `NumBad` data with group size indicated by `NumSample` appears in a **Graph** window.

RESAMPLE

In statistical analysis, the researcher is usually interested in obtaining not only a point estimate of a statistic, but also the variation in the point estimate, as well as confidence intervals for the true value of the parameter. For example, a researcher may calculate not only a sample mean, but also the standard error of the mean and a confidence interval for the mean.

The traditional methods for calculating standard errors and confidence intervals generally rely upon a statistic, or some known transformation of it, being asymptotically normally distributed. If this normality assumption does not hold, the traditional methods may be inaccurate. Resampling techniques such as the bootstrap and jackknife provide estimates of the standard error, confidence intervals, and distributions for any statistic. To use these procedures, you must supply the name of the data set under examination and an S-PLUS function or expression that calculates the statistic of interest.

Bootstrap Inference

In the *bootstrap*, a specified number of new samples are drawn by sampling with replacement from the data set of interest. The statistic of interest is calculated for each set of data, and the resulting set of estimates is used as an empirical distribution for the statistic.

Performing bootstrap inference

From the main menu, choose **Statistics ► Resample ► Bootstrap**. The **Bootstrap Inference** dialog opens, as shown in Figure 8.74.

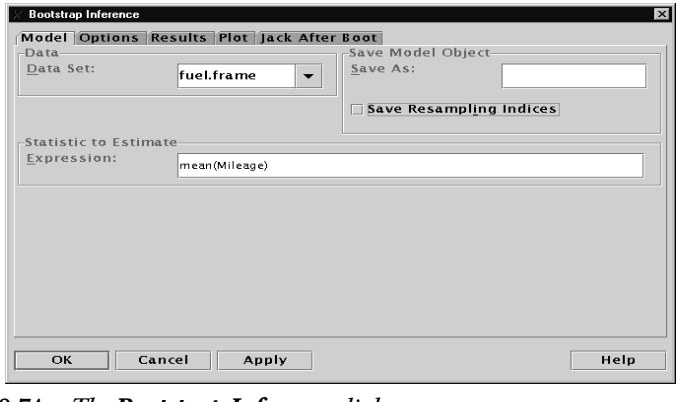


Figure 8.74: *The **Bootstrap Inference** dialog.*

Example 1

The data set `fuel.frame` is taken from the April 1990 issue of *Consumer Reports*. It contains 60 observations (rows) and 5 variables (columns). Observations of weight, engine displacement, mileage, type, and fuel were taken for each of sixty cars. We obtain bootstrap estimates of mean and variation for the mean of the `Mileage` variable.

1. Open the **Bootstrap Inference** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. Type `mean(Mileage)` in the **Expression** field.
4. On the **Options** page, type **250** in the **Number of Resamples** field to perform fewer than the default number of resamples. This speeds up the computations required for this example.
5. Click on the **Plot** page, and notice that the **Distribution of Replicates** plot is selected by default.
6. Click **OK**.

A bootstrap summary appears in the **Report** window, and a histogram with a density line is plotted in a **Graph** window.

Example 2

In this example, we obtain bootstrap estimates of mean and variation for the coefficients of a linear model. The model we use predicts `Mileage` from `Weight` and `Disp.` in the `fuel.frame` data set.

1. Open the **Bootstrap Inference** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. Type `coef(lm(Mileage ~ Weight+Disp., data=fuel.frame))` in the **Expression** field.
4. On the **Options** page, type **250** in the **Number of Resamples** field to perform fewer than the default number of resamples. This speeds up the computations required for this example.

5. Click on the **Plot** page, and notice that the **Distribution of Replicates** plot is selected by default.
6. Click **OK**.

A bootstrap summary appears in the **Report** window. In addition, three histograms with density lines (one for each coefficient) are plotted in a **Graph** window.

Jackknife Inference

In the jackknife, new samples are drawn by replicating the data, leaving out a single observation from each sample. The statistic of interest is calculated for each set of data, and this jackknife distribution is used to construct estimates.

Performing jackknife inference

From the main menu, choose **Statistics ► Resample ► Jackknife**. The **Jackknife Inference** dialog opens, as shown in Figure 8.75.

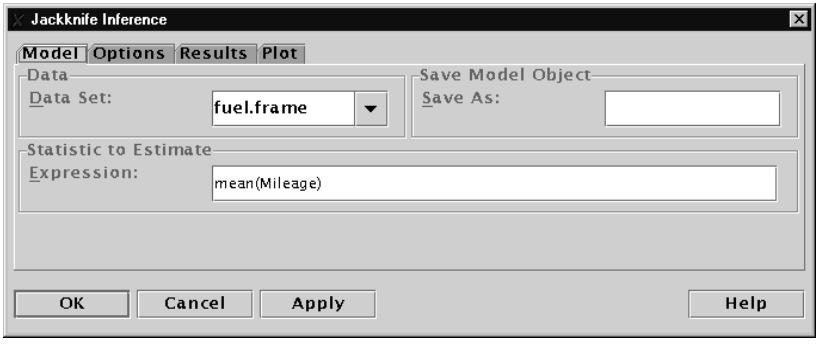


Figure 8.75: *The **Jackknife Inference** dialog.*

Example 1

We obtain jackknife estimates of mean and variation for the mean of `Mileage` in the `fuel.frame` data.

1. Open the **Jackknife Inference** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. Type `mean(Mileage)` in the **Expression** field.

4. Click on the **Plot** page, and notice that the **Distribution of Replicates** plot is selected by default.
5. Click **OK**.

A jackknife summary appears in the **Report** window, and a histogram with a density line is plotted in a **Graph** window.

Example 2

In this example, we obtain jackknife estimates of mean and variation for the coefficients of a linear model. The model we use predicts `Mileage` from `Weight` and `Disp.` in the `fuel.frame` data set.

1. Open the **Jackknife Inference** dialog.
2. Type `fuel.frame` in the **Data Set** field.
3. Type `coef(lm(Mileage ~ Weight+Disp., data=fuel.frame))` in the **Expression** field.
4. Click on the **Plot** page, and notice that the **Distribution of Replicates** plot is selected by default.
5. Click **OK**.

A jackknife summary appears in the **Report** window. In addition, three histograms with density lines (one for each coefficient) are plotted in a **Graph** window.

SMOOTHING

Smoothing techniques model a univariate response as a smooth function of a univariate predictor. With standard regression techniques, parametric functions are fit to scatter plot data. Frequently, you do not have enough prior information to determine what kind of parametric function to use. In such cases, you can fit a *nonparametric curve*, which does not assume a particular type of relationship.

Nonparametric curve fits are also called *smoothers* since they attempt to create a smooth curve showing the general trend in the data. The simplest smoothers use a *running average*, where the fit at a particular x value is calculated as a weighted average of the y values for nearby points. The weight given to each point decreases as the distance between its x value and the x value of interest increases. In the simplest kind of running average smoother, all points within a certain distance (or window) from the point of interest are weighted equally in the average for that point. The window width is called the *bandwidth* of the smoother, and is usually given as a percentage of the total number of data points. Increasing the bandwidth results in a smoother curve fit but may miss rapidly changing features. Decreasing the bandwidth allows the smoother to track rapidly changing features more accurately, but results in a rougher curve fit.

More sophisticated smoothers add variations to the running average approach. For example, smoothly decreasing weights or local linear fits may be used. However, all smoothers have some type of smoothness parameter (bandwidth) controlling the smoothness of the curve. The issue of good bandwidth selection is complicated and has been treated in many statistical research papers. You can, however, gain a good feeling for the practical consequences of varying the bandwidth by experimenting with smoothers on real data.

This section describes how to use four different types of smoothers.

- **Kernel Smoother**: a generalization of running averages in which different weight functions, or *kernels*, may be used. The weight functions provide transitions between points that are smoother than those in the simple running average approach.
- **Loess Smoother**: a noise-reduction approach that is based on local linear or quadratic fits to the data.

- **Spline Smoother**: a technique in which a sequence of polynomials is pieced together to obtain a smooth curve.
- **Supersmoother**: a highly automated variable span smoother. It obtains fitted values by taking weighted combinations of smoothers with varying bandwidths.

Kernel Smoother

A *kernel smoother* is a generalization of running averages in which different weight functions, or *kernels*, may be used. The weight functions provide transitions between points that are smoother than those in the simple running average approach. The default kernel is the *normal* or *Gaussian kernel*, in which the weights decrease with a Gaussian distribution away from the point of interest. Other choices include a triangle, a box, and the Parzen kernel. In a *triangle kernel*, the weights decrease linearly as the distance from the point of interest increases, so that the points on the edge of the smoothing window have a weight near zero. A *box* or *boxcar smoother* weighs each point within the smoothing window equally, and a *Parzen kernel* is a box convolved with a triangle.

Local Regression (Loess)

Local regression, or *loess*, was developed by W.S. Cleveland and others at Bell Laboratories. It is a clever approach to smoothing that is essentially a noise-reduction algorithm. Loess smoothing is based on local linear or quadratic fits to the data: at each point, a line or parabola is fit to the points within the smoothing window, and the predicted value is taken as the y value for the point of interest. Weighted least squares is used to compute the line or parabola in each window. Connecting the computed y values results in a smooth curve.

For loess smoothers, the bandwidth is referred to as the *span* of the smoother. The span is a number between 0 and 1, representing the percentage of points that should be included in the fit for a particular smoothing window. Smaller values result in less smoothing, and very small values close to 0 are not recommended. If the span is not specified, an appropriate value is computed using cross-validation. For small samples $(n < 50)$, or if there are substantial serial correlations between observations close in x value, a prespecified fixed span smoother should be used.

Spline Smoother

Spline smoothers are computed by piecing together a sequence of polynomials. Cubic splines are the most widely used in this class of smoothers, and involve locally cubic polynomials. The local polynomials are computed by minimizing a penalized residual sum of squares. Smoothness is assured by having the value, slope, and curvature of neighboring polynomials match at the points where they meet. Connecting the polynomials results in a smooth fit to the data. The more accurately a smoothing spline fits the data values, the rougher the curve, and vice versa.

The smoothing parameter for splines is called the *degrees of freedom*. The degrees of freedom controls the amount of curvature in the fit, and corresponds to the degree of the local polynomials. The lower the degrees of freedom, the smoother the curve. The degrees of freedom automatically determines the smoothing window, by governing the trade-off between smoothness of the fit and fidelity to the data values. For n data points, the degrees of freedom should be between 1 and $n-1$. Specifying $n-1$ degrees of freedom results in a curve that passes through each of the data points exactly. If the degrees of freedom is not specified, a parameter estimate is computed by crossvalidation.

Supersmoother

The *supersmoother* is a highly automated variable span smoother. It obtains fitted values by taking a weighted combination of smoothers with varying bandwidths. The smoothing parameter for supersmoothers is called the *span*. The span is a number between 0 and 1, representing the percentage of points that should be included in the fit for a particular smoothing window. Smaller values result in less smoothing, and very small values close to 0 are not recommended. If the span is not specified, an appropriate value is computed using crossvalidation. For small samples $(n < 50)$, or if there are substantial serial correlations between observations close in *x* value, a prespecified fixed span smoother should be used.

Examples

The `air` data set contains 111 observations (rows) and 4 variables (columns). It is taken from an environmental study that measured the four variables ozone, solar radiation, temperature, and wind speed for 111 consecutive days. We create smooth plots of `ozone` versus `radiation`.

1. Choose **Statistics ▶ Smoothing ▶ Kernel Smoother**. Select `air` as the **Data Set**, `radiation` as the **x Axis Value**, and `ozone` as the **y Axis Value**. Click **OK**. A **Graph** window is created containing a plot of `ozone` versus `radiation` with a kernel smooth.
2. Choose **Statistics ▶ Smoothing ▶ Loess Smoother**. Select `air` as the **Data Set**, `radiation` as the **x Axis Value**, and `ozone` as the **y Axis Value**. Click **OK**. A **Graph** window is created containing a plot of `ozone` versus `radiation` with a loess smooth.
3. Choose **Statistics ▶ Smoothing ▶ Spline Smoother**. Select `air` as the **Data Set**, `radiation` as the **x Axis Value**, and `ozone` as the **y Axis Value**. Click **OK**. A **Graph** window is created containing a plot of `ozone` versus `radiation` with a smoothing spline smooth.
4. Choose **Statistics ▶ Smoothing ▶ Supersmoother**. Select `air` as the **Data Set**, `radiation` as the **x Axis Value**, and `ozone` as the **y Axis Value**. Click **OK**. A **Graph** window is created containing a plot of `ozone` versus `radiation` with a supersmoother smooth.

TIME SERIES

Time series techniques are applied to sequential observations, such as daily measurements. In most statistical techniques, such as linear regression, the organization of observations (rows) in the data is irrelevant. In contrast, time series techniques look for correlations between neighboring observations.

This section discusses the time series available from the **Statistics ▶ Time Series** menu:

- **Autocorrelations**: calculates autocorrelations, autocovariances, or partial autocorrelations for sequential observations.
- **ARIMA**: fits autoregressive integrated moving average models to sequential observations. These are very general models that allow inclusion of autoregressive, moving average, and seasonal components.
- **Lag plot:** plots a time series versus lags of the time series.
- **Spectrum plot:** plots the results of a spectrum estimation.

We use these techniques to examine the structure in an environmental data set.

Autocorrelations

The *autocovariance function* is an important tool for describing the serial (or temporal) dependence structure of a univariate time series. It reflects how much correlation is present between lagged observations.

Plotting autocorrelations

From the main menu, choose **Statistics ▶ Time Series ▶ Autocorrelations**. The **Autocorrelations and Autocovariances** dialog opens, as shown in Figure 8.76.

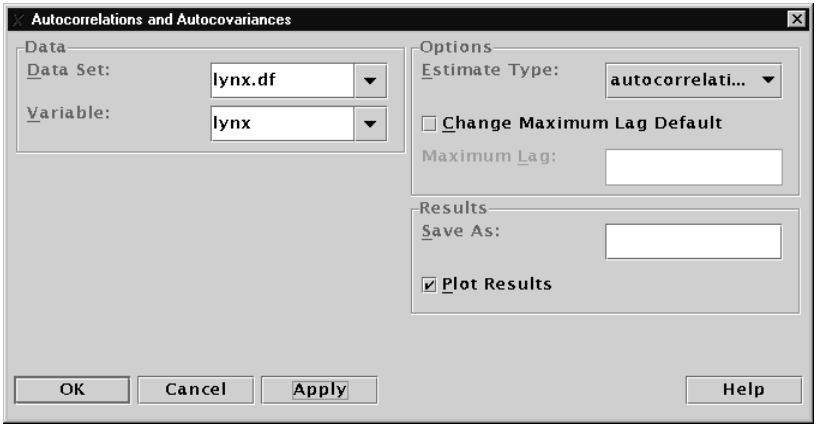


Figure 8.76: *The **Autocorrelations and Autocovariances** dialog.*

Example

The example data set `lynx` contains the annual number of lynx trappings in the Mackenzie River District of North-West Canada for the period 1821 to 1934. We can plot the data with the `ts.plot` command as follows:

```
> ts.plot(lynx, type="b", xlab="year", ylab="lynx", pch=1)
```

Figure 8.77 displays the graph.

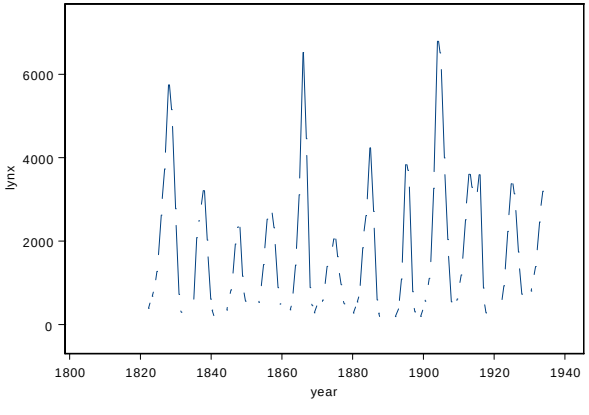


Figure 8.77: *Lynx trappings in the Mackenzie River District of North-West Canada.*

A definite cycle is present in the data. We can use autocorrelations to explore the length of the cycle.

By default, `lynx` is stored in an object of class `"ts"`. Before it can be recognized by the dialogs, we must store `lynx` as a column in a data frame. To do this, type the following in the **Commands** window:

```
> lynx.df <- data.frame(lynx)
```

We can now proceed with the autocorrelation analysis on the `lynx.df` data frame:

1. Open the **Autocorrelations and Autocovariances** dialog.
2. Type `lynx.df` in the **Data Set** field.
3. Select `lynx` as the **Variable**.
4. Click **OK**.

Figure 8.78 displays the resulting autocorrelation plot. The peaks at 10 and troughs at 5 reflect a ten-year cycle.

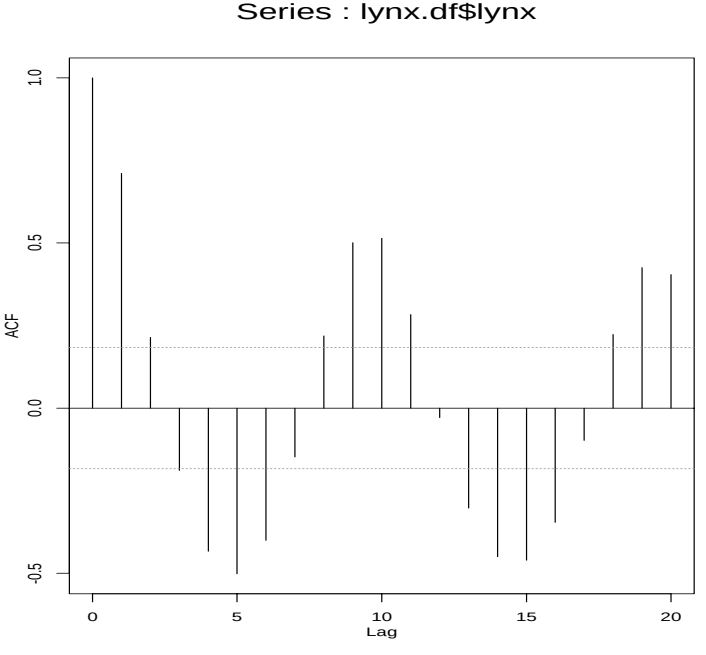


Figure 8.78: *Autocorrelation plot of the `lynx` data.*

ARIMA

Autoregressive integrated moving-average (ARIMA) models are useful for a wide variety of time series analyses, including forecasting, quality control, seasonal adjustment, and spectral estimation, as well as providing summaries of the data.

Fitting an ARIMA model

From the main menu, choose **Statistics ► Time Series ► ARIMA Models**. The **ARIMA Modeling** dialog opens, as shown in Figure 8.79.

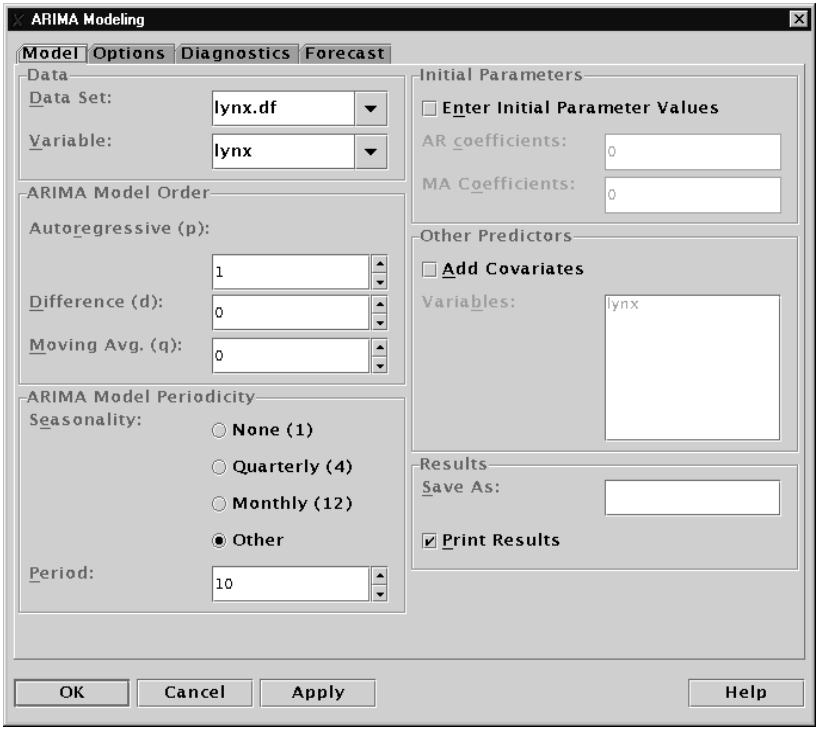


Figure 8.79: *The **ARIMA Modeling** dialog.*

Example

In the section Autocorrelations on page 421, we computed autocorrelations for the `lynx` time series. The autocorrelation plot in Figure 8.78 displays correlations between observations in the `lynx` data, with a ten-year cycle to the correlations. We can model this as an autoregressive model with a period of 10.

1. If you have not done so already, create the `lynx.df` data frame. The instructions for doing this are given on page 422.
2. Open the **ARIMA Modeling** dialog.
3. Type `lynx.df` in the **Data Set** field.
4. Select `lynx` as the **Variable**.
5. Specify an **Autoregressive Model Order** of 1.
6. Select **Other** as the **Seasonality**.
7. Specify a **Period** of 10.
8. Click **OK**.

Summaries for the ARIMA model are displayed in the **Report** window:

```
   ***  ARIMA Model Fitted to Series lynx.df$lynx ***

Method:  Maximum Likelihood
Model :  1 0 0
Period:  10

Coefficients:
  AR : 0.73883

Variance-Covariance Matrix:
            ar(10)
ar(10) 0.004366605

Optimizer has  converged
Convergence Type: relative function convergence
AIC: 1793.16261
```

Lag Plot

The **Lag Plot** dialog plots a time series versus lags of the time series.

Creating a lag plot

From the main menu, choose **Statistics ▶ Time Series ▶ Lag Plot**. The **Lag Plot** dialog opens, as shown in Figure 8.80.

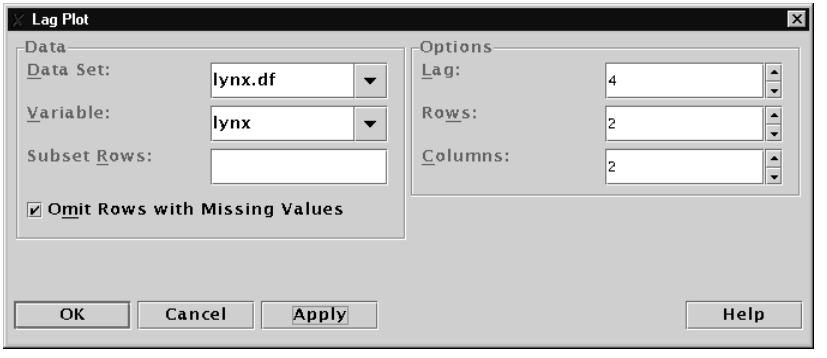


Figure 8.80: *The **Lag Plot** dialog.*

Example

In the section Autocorrelations on page 421, we computed autocorrelations for the `lynx` time series. In this example, we use a lag plot to example the correlation between observations at different lags.

1. If you have not done so already, create the `lynx.df` data frame with the instructions given on page 422.
2. Open the **Lag Plot** dialog.
3. Type `lynx.df` in the **Data Set** field.
4. Select `lynx` as the **Variable**.
5. Select a **Lag** of **4**.
6. Select a layout of **2 Rows** by **2 Columns**, and click **OK**.

A lag plot of the `lynx` data appears in a **Graph** window.

Spectrum Plot

The **Spectrum Plot** dialog plots the results of a spectral estimation. This plot displays the estimated spectrum for a time series using either a smoothed periodogram or autoregressive parameters.

Creating a spectrum plot

From the main menu, choose **Statistics ▶ Time Series ▶ Spectrum Plot**. The **Spectrum Plot** dialog opens, as shown in Figure 8.81.

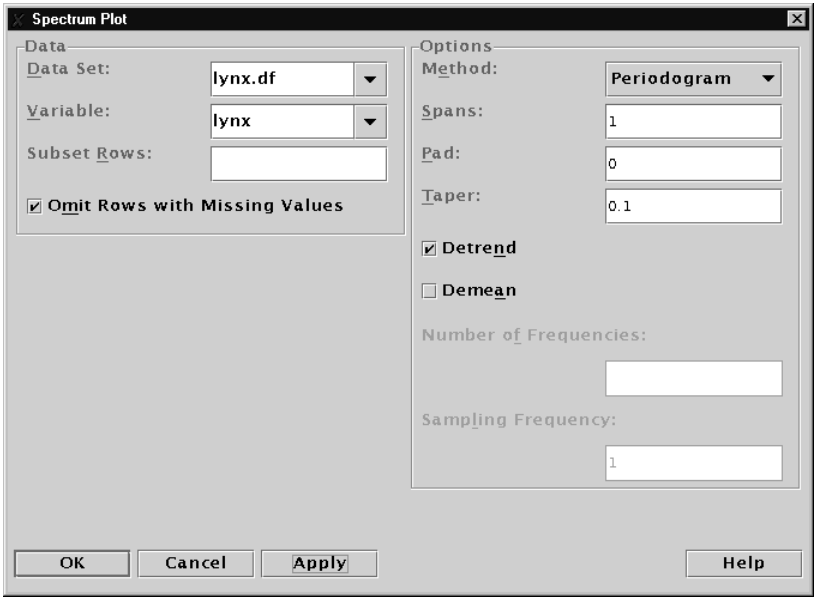


Figure 8.81: *The **Spectrum Plot** dialog.*

Example

In the section Autocorrelations on page 421, we computed autocorrelations for the `lynx` time series. In this example, we plot a smoothed periodogram of the `lynx` data to examine the periodicities in the series.

1. If you have not done so already, create the `lynx.df` data frame with the instructions given on page 422.
1. Open the **Spectrum Plot** dialog.
2. Type `lynx.df` in the **Data Set** field
3. Select `lynx` as the **Variable**, and click **OK**.

A spectrum plot of the `lynx` data appears in a **Graph** window.

REFERENCES

Box, G.E.P., Hunter, W.G., & Hunter, J.S. (1978). *Statistics for Experimenters*. New York: Wiley.

Chambers, J.M., Cleveland, W.S., Kleiner, B. & Tukey, P.A. (1983). *Graphical Methods for Data Analysis*. Belmont, California: Wadsworth.

Cleveland, W.S. (1979). Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74: 829-836.

Cleveland, W.S. (1985). *The Elements of Graphing Data*. Monterrey, California: Wadsworth.

Fleiss, J.L. (1981). *Statistical Methods for Rates and Proportions* (2nd ed.). New York: Wiley.

Friedman, J.H. (1984). *A Variable Span Smoother*. Technical Report No. 5, Laboratory for Computational Statistics. Department of Statistics, Stanford University, California.

Laird, N.M. & Ware, J.H. (1982). Random-Effects Models for Longitudinal Data. *Biometrics*, 38: 963-974.

Lindstrom, M.J. & Bates, D.M. (1990). Nonlinear Mixed Effects Models for Repeated Measures Data. *Biometrics*, 46: 673-687.

Snedecor, G.W. & Cochran, W.G. (1980). *Statistical Methods* (7th ed.). Ames, Iowa: Iowa State University Press.

Venables, W.N. & Ripley B.D. (1999). *Modern Applied Statistics with S-PLUS* (3rd ed.). New York: Springer.

CUSTOMIZING YOUR S-PLUS SESSION

9

Introduction 430

Setting S-PLUS Options 431

Setting Environment Variables 433

Customizing Your Session at Start-up and Closing 435
- Creating a **.S.chapters** File 436
- Creating a **.S.init** File 437
- Creating the `.First` Function 437
- Setting **S_FIRST** 437
- Customizing Your Session at Closing 438

Using Personal Function Libraries 439
- Creating an S Chapter 439
- Placing the Chapter in Your Search Path 440

Specifying Your Working Directory 441

Specifying a Pager 442

Environment Variables and `printgraph` 443

Setting Up Your Window System 445
- Setting X11 Resources 445
- S-PLUS X11 Resources 446
- Common Resources for the Motif Graphics Device 446

INTRODUCTION

S-PLUS offers a number of ways to customize your session. You can set options specifying how S-PLUS displays data and other information, create your own library of functions, or load C or Fortran code. You can even define a function to set these options each time you start S-PLUS, and another function to "clean up" each time you end a session.

This chapter describes changes that apply only to your S-PLUS session. To install them for every user on your system, talk with your system administrator or see the procedures in the *Installation and Maintenance Guide*.

SETTING S-PLUS OPTIONS

Options in S-PLUS serve much the same purpose as environment variables in UNIX–they determine the behavior of many aspects of the S-PLUS environment. You can set or modify these options with the `options` command. For example, to tell S-PLUS to echo back to the screen the commands you type in, use this expression:

```
> options(echo=T)
```

Table 9.1 lists some of the most useful options you can set. See the `options` help file for a complete description of the available options. If you want to set an option each time you start a session, see the section Customizing Your Session at Start-up and Closing (page 435).

You can also determine the value of any option with `options`. For example, to find the current value of the `echo` option, type the following expression at the `>` prompt:

```
> options("echo")
```

S-PLUS answers with the following:

```
options("echo")
$echo:
[1] T
```

Because `echo` is true (we set it in the first paragraph of this section), S-PLUS prints the command you type in before returning the requested value.

Table 9.1: *Some of the options available with the `options` function.*

`echo`	tells S-PLUS whether to repeat commands it receives back to the screen. The default value is `echo=F`.
`prompt`	tells S-PLUS what character string to print when it is ready for input. The default value is `prompt=">"`.
`continue`	tells S-PLUS which character string to print when you press the return key before completing an S-PLUS expression. The default value is `continue="+ "`.

Table 9.1: *Some of the options available with the* `options` *function.*

`width`	tells S-PLUS how wide the screen is. You can change this value to get the `print` command to create very wide or very narrow lines. The default value is `width=80`.
`length`	tells S-PLUS how tall the screen is. This controls how frequently the `print` command prints out the summary of column names when printing a matrix. The default value is `length=48`.
`check`	tells S-PLUS to perform automatic validity checking at various points in the evaluation. The default is false, or `check=F`.
`editor`	tells S-PLUS what text editor will be used in `history` and `fix`. The default is `vi`.
`digits`	tells many of the printing functions how many digits to use when printing numbers. The default value is `digits=7`.
`pager`	tells S-PLUS what pager program to use in such places as the `help` and `page` functions. The default for `pager` is the value of environment variable **S_PAGER**, which in turn defaults to the value of environment variable **PAGER**, or **"less"** if that is not set.

SETTING ENVIRONMENT VARIABLES

Table 9.2 is a list of the environment variables recognized by S-PLUS. You are *not* required to set them.

Many of the variables in this section take effect if you set them to any value, and do not take effect if you do not set them, so you may leave them unset without harm. For example, to set **S_SILENT_STARTUP** you can enter:

```
setenv S_SILENT_STARTUP X
```

on the command line and S-PLUS will not print its copyright information on start-up, because the variable **S_SILENT_STARTUP** has a value (any value).

User code can check the current values for these variables by using `getenv` from C or S code.

Table 9.2: *Environment variables recognized by* S-PLUS.

Variable	Description
ALWAYS_PROMPT	Chiefly affects the actions of the `parse` function. Normally, `parse` prompts for input only when the input appears to be coming from a terminal. When **ALWAYS_PROMPT** is set (to anything at all), `parse` prompts even if the standard input and standard error streams are pipes or files. See the `parse` help file for more details.
EDITOR	Sets the command line editor to either **emacs** or **vi**. Overridden by **S_CLEDITOR** or **VISUAL** if either contains a valid value.
PATH	Specifies the directories which are searched when a command is issued to the UNIX shell. In particular, the **Splus5** command should be installed in one of the listed directories.
S_CLEDITOR	Sets the command line editor to either **emacs** or **vi**.
S_CLHISTFILE	Sets the name of the command line editor's history file. The default is **$HOME/.Splus_history** .
S_CLHISTSIZE	Specifies the maximum number of lines to put in the command line editor's history file.
S_CLNOHIST	Suppresses writing of the command line editor's history file.
S_EDITOR	Sets the value of `options()$editor`. The specified editor is used by the `fix` function.

Table 9.2: *Environment variables recognized by* S-PLUS.

S_FIRST	S-PLUS function evaluated at start-up. See section Setting S_FIRST (page 437).
SHELL	Specifies the UNIX command shell, which S-PLUS uses to determine the shell to use in shell escapes (`!`) if **S_SHELL** is not set.
SHOME	Specifies the directory where S-PLUS is installed. By default, this is set to the parent directory of the program executable.
S_PAGER	Specifies which pager to use. Sets the value of `options()$pager`; the specified pager is used by the `page`, `help`, and `?` functions.
S_POSTSCRIPT_PRINT_COMMAND	Specifies the UNIX command (**lp**, **lpr**, etc.) used to send files to a PostScript printer.
S_PRINTGRAPH_ONEFILE	Determines whether plots generated by the `postscript` function are accumulated in a single file (`TRUE`) or whether each plot is put in a separate EPS file. This environment variable sets the default for the `onefile` arguments to `ps.options` and `postscript`.
S_PRINT_ORIENTATION	Specifies the orientation of the graphic as **landscape** or **portrait**. Determines the default value of the `horizontal` argument to `ps.options` and `printgraph`.
S_SHELL	Specifies the shell used during shell escapes, that is, commands issued from the escape character (`!`). The default value is the value of **SHELL**.
S_SILENT_STARTUP	Disable printing of copyright/version messages.
S_WORK	Specifies the location of the *working* data directory, that is, the directory in which S-PLUS creates and reads data objects.
VISUAL	Sets the command line editor to either **emacs** or **vi**. Overridden by **S_CLEDITOR** if it contains a valid value.

CUSTOMIZING YOUR SESSION AT START-UP AND CLOSING

If you routinely set one or more options each time you start S-PLUS, or want to automatically attach library sections or S-PLUS chapters, you can store these choices and have S-PLUS set them automatically whenever it starts.

When you start S-PLUS, the following initialization steps occur:

1. Basic initialization brings the evaluator to the point of being able to evaluate expressions.
2. S-PLUS then looks for the standard initialization file **$SHOME/S.init**. This is a text file containing S-PLUS expressions. The default initialization file performs the remaining steps in this list.
3. If your system administrator has performed any site customization in the file **$SHOME/local/S.init**, the actions in that file are evaluated next.
4. S-PLUS next looks for the file **$SHOME/.S.chapters** , which is a text file containing paths of library sections or S-PLUS chapters to be attached for all users. By default, this file does not exist, since only the standard S-PLUS libraries are attached during the basic initialization.
5. S-PLUS next looks for your personal **.S.chapters** file, first in the current directory, and then if not found, in your **MySwork** directory. You should list in this file any library sections or S-PLUS chapters you want attached at start-up.
6. S-PLUS then determines your working data; see the section Specifying Your Working Directory for details.
7. S-PLUS evaluates the customization file **.S.init** if it is found in either the current directory or your **MySwork** directory. The **.S.init** file is a text file containing S-PLUS expressions that are executed at the start of your session. Note that this file is different than **$SHOME/S.init** , which affects all users' sessions.
8. S-PLUS evaluates the function `.First.Sys`, which includes evaluating the local system initialization function `.First.local` if it exists.

9. S-PLUS evaluates the environment variable **S_FIRST** , if set, or the first `.First` function found in the search paths set by steps 3–5.

In most cases, the initialization process includes only one of steps 6 and 8 above. Thus, you will probably use only one of the following mechanisms to set your start-up options:

- Create an S-PLUS function named `.First` containing the desired options.
- Create a text file of S-PLUS tasks named **.S.init** in either your current directory or your **MySwork** directory.
- Set the S-PLUS environment variable **S_FIRST** as described below.

The `.First` function is the traditional S-PLUS initialization tool. The **.S.init** file has the advantage of being a text file that can easily be edited outside of S-PLUS. The **S_FIRST** variable is a convenient way to override `.First` for a specific S-PLUS session.

Creating a .S.chapters File

If you want to attach specific S-PLUS chapters or library sections in your S-PLUS session, you can specify those directories using a **.S.chapters** file. Here is a sample **.S.chapters** file that attaches a specific users utility functions and also the `maps` library:

```
/homes/rich/Sstuff/utilities
maps
```

Paths beginning in "`/`" (including those using environment variables that evaluate to a path beginning in "`/`") are interpreted as absolute paths, those that begin with any other character are interpreted as paths relative to **$SHOME/library** .

You can create a **.S.chapters** file in any directory in which you want to start-up S-PLUS. S-PLUS checks both the current directory and the default S-PLUS start-up directory **MySwork** to see whether this initialization file exists, and evaluates the first one it finds.

Creating a .S.init File

Here is a sample **.S.init** file that sets the output width for the session as well as the default displayed precision:

```
{
  options(width=55, digits=4)
}
```

You can create a **.S.init** file in any directory in which you want to start-up S-PLUS. S-PLUS checks both the current directory and the default S-PLUS start-up directory **MySwork** to see whether this initialization file exists, and evaluates the first one it finds.

Creating the .First Function

Here is a sample `.First` function that starts the Motif graphics device:

```
> .First <- function() motif()
```

After creating a `.First` function, you should always test it immediately to make sure it works. Otherwise S-PLUS will not execute it in subsequent sessions.

Setting S_FIRST

To store a sequence of commands in the **S_FIRST** variable, use the following syntax:

```
setenv S_FIRST 'S-PLUS expression' # C shell
set S_FIRST= 'S-PLUS expression'; export S_FIRST  # Bourne or
                                                  # Korn shell
```

For example, the following C shell command tells S-PLUS to start the Motif graphics device:

```
setenv S_FIRST 'motif()'
```

To avoid misinterpretation by the command line parser, it is safest to surround complex S-PLUS expressions with either single or double quotes (whichever you do *not* use in your S-PLUS expression).

You can also combine several commands into a single S-PLUS function, then set **S_FIRST** to this function. For example:

```
> startup <- function() { options(digits=4)
+ options(expressions=128)}
```

You can call this function each time you start S-PLUS by setting **S_FIRST** as follows:

```
setenv S_FIRST 'startup()'
```

Variables can only be defined at initialization, and not while S-PLUS is running. Any changes to **S_FIRST** will take effect only upon restarting S-PLUS.

Customizing Your Session at Closing

When S-PLUS quits, it looks in your data directory for a function called `.Last`. If `.Last` exists, S-PLUS runs it. A `.Last` function can be useful for cleaning up your directory by removing temporary objects or files.

USING PERSONAL FUNCTION LIBRARIES

If you write functions that you want to use many times, you should not store them in your working directory, because objects in this directory are easily overwritten. Instead, to prevent yourself from inadvertently removing your functions, you should create a personal function library to hold them. A personal function library is simply an S chapter that you add to your S-PLUS search path, allowing you to access your functions from wherever you start S-PLUS.

If you are working on a number of different projects, you can create personal function libraries for each project to store the functions developed for that project.

To set up your own library, there are two main steps:

1. Create an S chapter to hold your library of functions and help files.
2. Place the new directory in your S-PLUS search path.

We describe these steps in detail in the following subsections.

Note

If your function library would be useful to many people on your system, you can ask your system administrator to create a system-wide version of your function library that everyone can access with the S-PLUS `library` function.

Creating an S Chapter

To create a chapter, you use the UNIX **mkdir** command from the UNIX prompt, followed by the S-PLUS utility **CHAPTER**. For example, to create an S-PLUS chapter called **mysplus** in your home directory, use the following commands:

```
% cd
% mkdir mysplus
% cd mysplus
% Splus CHAPTER
```

The **Splus CHAPTER** utility creates a **.Data** directory in the directory you created with **mkdir**; you will store your functions in this **.Data** subdirectory. The **.Data** subdirectory is created with two subdirectories, **__Help** and **__Meta**, which are used to store help files and object metadata, respectively.

Note
You can create your S chapter directory anywhere you have write permission, and you can name it anything you like.

Placing the Chapter in Your Search Path

To add an S chapter to your search path, use the S-PLUS `attach` function, which provides temporary access to a directory during an S-PLUS session. You name the directory to be added as a character-string argument to `attach`. For example, to add the chapter **/usr/rich/mysplus** to your search path with `attach`, use the following expression:

```
> attach("/usr/rich/mysplus")
```

When specifying directories to `attach`, you must specify the complete path name. S-PLUS does not expand such UNIX conventions as **~bob** or **$HOME**.

Any directories you attach are detached when you quit S-PLUS. In order to have your functions available at all times, you can specify the chapter as part of your **.S.chapters** file:

```
...other attached files
/spud/users/mysplus
...other attached files
```

You can also use either the **S.init** file or a `.First` function to attach `mysplus` to your S-PLUS search list, as in the following example:

```
> .First <- function(){
+ attach("/spud/users/mysplus")
+ }
```

Whenever you start S-PLUS, `mysplus` is automatically attached, and your functions and help files are made available.

SPECIFYING YOUR WORKING DIRECTORY

Whenever you assign the results of an S-PLUS expression to an object, using the <- or = operator within an S-PLUS session, S-PLUS creates the named object in your *working directory*. The working directory occupies position 1 in your S-PLUS search list, so it is also the first place S-PLUS looks for an S-PLUS object.

You specify the working directory with the environment variable **S_WORK**, which can specify one directory or a colon-separated list of directories. The first valid directory in the list is used as the working directory, and the others are placed behind it in the search list. To be valid, a directory must be a valid S-PLUS chapter and be one for which you have write permission. If **S_WORK** is set but contains no valid S-PLUS chapters, attempting to launch S-PLUS results in an error.

For example, to specify the chapter **/usr/rich/mysplus** as your working directory, set **S_WORK** as follows:

```
setenv S_WORK /usr/rich/mysplus
```

If **S_WORK** is not set, S-PLUS sets the working directory as follows:

1. If the current directory is a valid S-PLUS 6 chapter, S-PLUS uses it as the working data.
2. Check for the existence of the directory **$HOME/MySwork** .
   - If it exists and is a valid S-PLUS 6 chapter, S-PLUS uses it as the working data.
   - If it exists but is not a valid S-PLUS 6 chapter, S-PLUS prints a warning, then creates a directory in **$HOME** with a name of the form **Schapter$$** , where **$$** is a number that guarantees the uniqueness of the chapter name, to use as the working data.
   - If it does not exist, S-PLUS creates it and initializes it as an S-PLUS 6 chapter, then uses it as the working data.

SPECIFYING A PAGER

A pager is a tool for viewing objects and files that are larger than can fit on your screen. They function much like editors for moving around files, but typically do not have actual editing functions. The most common uses for pagers in S-PLUS are to look at lengthy functions and data sets with the `page` function and to look at help files with the `help` function. The `page` function uses the pager specified in `options()$pager`, while the `help` function uses the pager specified in `options()$help.pager`.

The value of `options()$pager` is initially specified by the **S_PAGER** environment variable, if set, or to `"less"`, if not. You can use the `options` function to specify a new default pager at any time during your S-PLUS session. Modifications to **S_PAGER**, however, take effect only when you next start S-PLUS.

Using `options`, usually in your `.First` function, is the preferred method for setting your pager. Simply use the following function call:

```
> options(pager=pager)
```

where `pager` is a character string containing the command, with any necessary flags, used to start the pager.

The value of `options()$help.pager` defaults to `"slynx"`, which is a version of the **lynx** terminal-based Web browser. The help pager is used to display HTML text in a terminal window, as opposed to the JavaHelp window available via the `help.start()` command. Your help pager should therefore be an HTML-aware viewer, such as the default **slynx** browser. For more details, see the section Getting Help in S-Plus on page 21.

ENVIRONMENT VARIABLES AND PRINTGRAPH

S-PLUS uses environment variables to set defaults for the `printgraph` function. Your system administrator already set these variables system-wide, but if you would like to change the default values for your S-PLUS session, use your UNIX shell command to set a new value for the environment variable before you start S-PLUS.

Note

The `printgraph` function sets its defaults differently from the defaults for the *Print* button on graphics devices such as `motif`.

For example, to make `printgraph` produce plots with the *x*-axis on the short side of the paper, type the following from the C shell:

```
setenv S_PRINT_ORIENTATION portrait
```

Start S-PLUS. Any plots made with `printgraph` are now produced in portrait mode.

S-PLUS uses the following environment variables with `printgraph`:

- **S_PRINT_ORIENTATION** controls the orientation of plots. It has two possible values: “portrait", which puts the *x*-axis along the short side of the paper, and “landscape", which puts the *y*-axis along the short side of the paper.
- **S_PRINTGRAPH_ONEFILE** controls whether S-PLUS writes `printgraph` output to one file or many. It has two possible values: “yes" and “no". If “yes", `printgraph` sends its output to **PostScript.out** . If “no", `printgraph` creates a separate file each time and tries to send it to the printer by executing the command specified in the variable **S_POSTSCRIPT_PRINT_COMMAND** .
- **S_POSTSCRIPT_PRINT_COMMAND** sets the UNIX PostScript printing command.

Note

You cannot change the values of any environment variable once you start S-PLUS. If you want to change a variable, you must stop S-PLUS, change the variable, then start S-PLUS again. To change `printgraph`'s behavior temporarily, see the `printgraph` help file for optional arguments.

You can also modify `printgraph`'s behavior using options passed to `ps.options.send`. See the section Printing with PostScript Printers for details on how to control PostScript options.

SETTING UP YOUR WINDOW SYSTEM

The `motif` graphics device has a control panel to help you pick the colors, fonts, and printing commands you want for your S-PLUS graphics. When you save these settings, they are used each time you start one of these devices. You can also specify settings for these graphics devices by setting *X11 resources*.

The `motif` graphics device uses resources of the X Window System, Version 11, or X11. This section describes how to customize your graphics windows by setting X11 resources.

Setting X11 Resources

There are a number of ways you can set resources for X11 applications. You should talk with your system administrator about the way that is preferred on your system. This section describes one of the most flexible methods of setting X11 resources–using the **xrdb** command.

As with other X11 programs, before you can run the **xrdb** command, you must give it permission to access your display To do this, you need to first specify your *display server*, which controls the access to your display, and then explicitly give access to that server to the host on which you run **xrdb**. If you are running the C-shell, the network name of the computer or terminal you are sitting at is ***displayserver***, and the network name of the machine on which you run **xrdb** is ***remotehost***, you can give the appropriate permission with the following commands:

```
setenv DISPLAY displayserver:0
xhost + remotehost
```

The **setenv** command sets the **DISPLAY** environment variable to your window server so that every X11 program knows where to create windows. The **xhost** command gives the specified computer permission to create a window on your display.

The **xrdb** command takes a file of X11 resources as its argument and creates an *X11 Resource Database*. Whenever any X11 program tries to create a window on your display, the program first looks at your X11 resource data base to get default values. The **xrdb** command uses the C-preprocessor to set the defaults that are appropriate for your machine. See the **xrdb** manual page for more information.

S-PLUS X11 Resources

The file **SPlusMotif** in the directory **$SHOME/splus/lib/X11/app-defaults** holds the system-wide default values for the `motif` graphics device. Many of the resources declared in the defaults file are discussed below.

When you specify a resource use the form:

resource* : *value

where ***resource*** is the name of the resource you want to use and ***value*** is the value you want to give it. For example, set the resource which tells **xterm** windows to have a scrollbar with this command:

xterm*scrollBar : True

When you add this resource to your X11 resource data base, then create another window with the UNIX **xterm** command, the window has a scroll bar. In this example the name of the application for which you set defaults is **xterm**. When you want to set resources for your `motif` devices, you must use the proper application name, **sgraphMotif**.

For example, if you put the following resource into your resource data base:

sgraphMotif*copyScale : 0.75

you would specify the ratio of the size of your original graph to the size of any copies you created from it . When you create a copy of your `motif` graphics device, the copy is three-fourths the size of your current S-PLUS graphics window.

Common Resources for the Motif Graphics Device

The following resources are commonly used with the `motif` graphics device:

- **sgraphMotif*copyScale** sets the size ratio of the copy you produce when you click on the "Copy Graph" button. S-PLUS multiplies the height and the width of the canvas by the value in the **copyScale** resource to create the dimensions for the new window. The default resource declaration produces a copy with dimensions one half those of the current window:

  sgraphMotif*copyScale : 0.5

- **sgraphMotif*fonts** sets the fonts that the `motif` graphics device use for creating axis labels and plotting characters. The fonts must be named in order from smallest to largest. Use the UNIX command **xlsfonts** to see a complete list of the fonts available on your screen. As an example, the following resources tells the `motif` graphics device to use the **vg** family of fonts ranging in point size from 13 to 40:

  sgraphMotif*fonts : vg-13 vg-20 vg-25 vg-31 vg-40

Note

If you select names that are too long to fit on one line, use multiple lines, and make sure that each line but the last ends with a backslash (\). Since these fonts are intended to list available sizes of the same font, the actual font used is controlled by the current value of `par()$cex` and the size of the fonts relative to the **defaultFont** described below.

- **sgraphMotif*defaultFont** tells the `motif` graphics device which font in the ***font** resource list to use as the default font, when `cex=1`.

Note

The fonts are numbered from 0, so that the following resource tells the `motif` graphics devices to use the third font in the list given by **sgraphMotif*fonts** : **sgraphMotif*defaultFont : 2**

- **sgraphMotif*canvas.width** and **sgraphMotif*canvas.height** control the starting size of the drawing area of the graphics windows. The following resources set the size of the plotting area for the `motif` graphics device to 800 by 632 pixels.

  sgraphMotif*canvas.width : 800

  sgraphMotif*canvas.height : 632

Note

When S-PLUS creates graphics to display in the graphics windows, it uses the initial values of ***canvas.width** and ***canvas.height** resources as the size of the drawing area. If you create a graphics device with a small drawing area and later resize the graphics window to a larger size, the resolution of the graphics image is reduced, so that your plots may look "blocky."

To set color resources for `motif` devices interactively, we recommend that you use the menus provided in the graphics windows. You can also use the **sgraphMotif*colorSchemes** resource to define new color schemes. However, if you use **sgraphMotif*colorSchemes** to define new color schemes, you must copy the existing resource completely before defining your new schemes, or the old color schemes will be unavailable.

APPENDIX: MIGRATING FROM S-PLUS 3.4

If you are migrating from S-PLUS 3.4 or earlier to S-PLUS 6, use this appendix to help you make the most of your existing code. You will find that most everything you have done before will work as before. This section should describe the most baffling changes, as well as give you complete details on how to modify your existing work to take advantage of the many new features of S-PLUS 5.x and later.

The known incompatibilities between S-PLUS 3.4 and S-PLUS 5.x (and later) are listed below. Many of these incompatibilities are discussed in greater detail in the remaining subsections of this migration appendix.

- New binary data format
- New help file format
- Changes in assignment order and immediacy
- Changes in using compiled code
- New object-oriented programming model
- New interactive debugging tool
- Changes in data frame construction and coercion
- Loops modified to have no return values

Many of these changes arise from the change of the base S language to S Version 4, as described in the *S-PLUS Programmer's Guide* and the book by John M. Chambers, *Programming with Data*.

Converting S-PLUS 3.x Functions and Data

S-PLUS 5.x and later stores data in a new binary format which is not recognized by earlier versions of S-PLUS (although S-PLUS 5.x and later recognizes S-PLUS 3.x binary data). If you have old functions or data that you want to use with S-PLUS 6, you should convert your data to the new data format. The conversion process creates new copies of your functions and data sets, while preserving your existing data in its current format, leaving it available for use with S-PLUS 3.x.

To convert your data, use the following procedure. In this example, we assume your S-PLUS 3.x data is in a **.Data** directory under the directory **$HOME/mydata**:

1. Create a default S-PLUS 6 chapter (**my6xdata**), and an S-PLUS chapter for your converted data (**my34data**):

   mkdir my6xdata my34data
   cd my6xdata
   Splus CHAPTER
   cd ../my34data
   Splus CHAPTER
   cd ../my6xdata

2. Start S-PLUS 6 in your default S-PLUS 6 chapter:

   Splus

3. Call the function `convertOldLibrary`:

```
> convertOldLibrary(paste(getenv("HOME"),
+ "mydata", sep = "/"),
+ to = paste(getenv("HOME"), "my34data", sep="/"))
```

4. The converted data are now under **my34data**. You can now freely use this data with S-PLUS 6. The **mydata** directory is unchanged, so you can continue to use it with S-PLUS 3.4.

The conversion utility makes the following changes to your old objects:

1. Changes calls to the `class` function to call `oldClass`.
2. Change calls to the `log` function (which no longer accepts more than one argument) to call `logb` (which still accepts the `base` argument).
3. Change calls to `unclass` to call `oldUnclass`.
4. Creates metadata for old-style classes by generating calls to `setOldClass`. The primary use of this step is to allow inheritance in old-style classes. S-PLUS 3.x classes could be character vectors of any length, with the first element giving the actual class and additional elements showing classes from

which the class inherited. Thus, ordered factors had class `c("ordered", "factor")`. This inheritance path is preserved by using `setOldClass` to create the appropriate metadata.

There is also a new S-PLUS utility, **CONVERTOLDSCRIPTS**, that will convert **.q** files or other S-PLUS source files into files suitable for use with S-PLUS 5.x and later. This utility performs the same first two steps as `convertOldLibrary`; it does not currently convert class information. As described in the section Migrating Object-Oriented Program Code (page 458), updating class information can be problematic, especially if multiple inheritance is involved.

Neither `convertOldLibrary` nor **CONVERTOLDSCRIPTS** produce flawless code for S-PLUS 5.x and later; certain new requirements, such as that objects be locally assigned before replacement operations can be performed, are difficult to check for automatically. We *strongly* encourage you to examine all functions modified by either of these utilities to ensure they continue to do the appropriate actions.

Converting S-PLUS 3.x Help Files

According to *Programming with Data*'s Appendix B, `convertOldLibrary` will also convert old-style **nroff/troff** help files to documentation objects. S-PLUS 6 does not use documentation objects, but instead a new generation of help files formatted using SGML. The `prompt` function and its methods have been modified to produce these new help files.

If you have help files in your S-PLUS 3.4 working directory under **.Data/.Help**, you can convert them as follows:

```
> convertOldDoc(from = paste(getenv("HOME"),
+ "mydata", sep = "/"),
+ to = paste(getenv("HOME"), "my34data", sep="/"))
```

Changes in Assignment

For the most part, assignments work as they did in S-PLUS 3.4, with some significant changes. You may or may not experience any effects from these changes in your normal use of S-PLUS, but you should be aware of them in case you notice seemingly anomalous behavior.

New Assignment Operator

The = operator can now be used for assignment, as in C. The old-style assignment operator <- is still available, and remains the preferred assignment operator in S-PLUS, both because it is more suggestive of the fundamental asymmetry of the assignment operation, and because it avoids the risk of confusion with named arguments. To see how confusion might arise, suppose you are want to draw a sample from a generated run of random numbers, and you want to store the full run for possible later use. You might try (with the new = operator) something like the following:

```
> sample(my.samp = runif(400), 30)
```

only to see the following error message:

```
Problem in sample: argument my.samp= not matched: ...
```

If you try to save in the same way to the name `x`, the sample is drawn correctly, but `x` does not get assigned the results of `runif`. You can do what you want with the <- operator; the expression `sample(my.samp <- runif(400), 30)` both creates the object `my.samp` and draws the requested sample of size 30.

New Default for the immediate Argument

The `immediate` argument to `assign` now defaults to `TRUE` when the `where` argument is supplied. In addition, it is always `TRUE` if `where` is supplied and `where` is not the working data.

Changes in Commitment Order

As always in S-PLUS, assignments to and removals from permanent databases are usually committed to the database only at the end of the current top-level expression. In earlier versions of S-PLUS, however, assignments and removals were committed in the order in which they occurred in the evaluation of the top-level expression. In S-PLUS 6, all assignments are committed in their natural order, and then all removals are performed. This can generate spurious warnings about objects not found.

For example, consider the following not-very-useful function:

```
> test1 <- function()  {
+     assign("a", 1:10)
+     print(get("a"))
+     remove("a")
+     assign("a", 2:20)
+     print(get("a"))
+     remove("a")
+ }
```

When this function is called as a top-level expression, you get the following warning message:

```
object "a" to be removed, but not found in database "." in:
remove("a")
```

All of the assignments and removals are queued up, and so there are two assignments to `a`, and two removals of `a`. When the top-level expression completes, the assignments are committed, and then the removals are performed. The first removal rids the database of `a`, then generates the warning when the second removal cannot find `a`.

Migrating C and Fortran Code

Dynamic Linking

Dynamic loading (the `dyn.load` function) and static loading (the **LOAD** utility) are no longer supported. Compiled code is now added to S-PLUS by means of dynamic linking, using the **CHAPTER** mechanism described in *Programming with Data*. In most cases, this will be far simpler than the old compile-load routine. The best part is that when compiled code is needed for use with a library, the code is loaded automatically when the library is attached.

Suppose you have two existing compiled routines, one a C routine in a file named **myccode.c** and the second a Fortran routine in a file named **myfcode.f**. To use the new mechanism on your old C and Fortran code, create a new S-PLUS 6 chapter as follows:

mkdir mychapter

cp myccode.c myfcode.f mychapter

cd mychapter

Splus CHAPTER

The **CHAPTER** utility automatically creates a makefile with your source code and appropriate targets for compiling your routines and creating a shared object **S.so**. To create the shared object, use the following command:

Splus make

When you start S-PLUS 6 in this chapter, or whenever you attach this chapter to a running S-PLUS session, S-PLUS will automatically dynamically link the file **S.so** into the session, and your C and Fortran routines will be available. You no longer need to worry about `.First.lib` files, or `library.dynam`, or remembering to call `dyn.load`.

On occasion, you may want to dynamically link code that is not associated with an S chapter. You can do this with the `dyn.open` function, which replaces much of the functionality of the `dyn.load.shared` function. Routines linked with `dyn.open` can be unlinked using `dyn.close`. The `dyn.exists` function can be used to test for the availability of routines.

Changes to the .C and .Fortran Functions

The `.C` function has lost an argument and both `.C` and `.Fortran` have gained two new ones:

- The `pointers` argument is now absent from `.C`. Compiled code that uses the `pointers` argument will crash S-PLUS. Much of the functionality that the `pointers` argument was intended to support is now supported by the `.Call` function, which allows you to manipulate arbitrary S objects within C.
- The `COPY` argument controls copying of data; if you know your C or Fortran routine will not modify an argument, you can specify that argument not be copied.
- The `CLASSES` argument is used to ensure that arguments passed to `.C` and `.Fortran` are of the proper class. You can use this argument in place of explicit coercion calls such as those you might have in your existing C code. For example, here is an example from version 3.2 of the *S-PLUS Programmer's Manual*:

```
> my.norm <-
+ function(n)
+ {
+       .C("my_rnorm",
+             double(n),
+             as.integer(n))[[1]]
+ }
```

Using the new `CLASSES` argument, we can rewrite `my.norm` as follows:

```
> my.norm <-
+ function(n)
+ {
+       .C("my_rnorm", double(n), n,
+             CLASSES=c("double", "integer"))[[1]]
+ }
```

Note that the call to `double` is not a coercion; it is generating a double-precision vector of length `n`.

You can also use the new function `setInterface` to place the copy and classes information into the metadata.

The long-standing prohibition against Fortran I/O statements has been relaxed; Fortran `read` and `write` statements now work correctly.

Changed Arguments to Built-in Routines

If you use any routines distributed with S-PLUS (that is, if your code includes the line `#include <S.h>`), you may have to modify your calls to those routines. In particular, most of the calls now require the use of a new macro, `S_EVALUATOR`, and an additional argument, `S_evaluator`.

For example, version 3.2 of the *S-PLUS Programmer's Manual* includes the following example code:

```
#include <S.h>
my_rnorm(x, n_p)
double *x ;
long *n_p ;
{
        long i, n = *n_p ;
        seed_in((long *)NULL) ;
        for (i=0; i<n; i++)
            x[i] = norm_rand() ;
        seed_out((long *)NULL ;
}
```

The following code updates the example to ANSI C, and demonstrates the use of the `S_evaluator` argument:

```
#include "S.h"
void my_rnorm(double *x, long *n_p)
{
        S_EVALUATOR
        long i, n = *n_p ;
        seed_in((long *)NULL, S_evaluator) ;
        for (i=0 ; i<n ; i++)
                x[i] = norm_rand(S_evaluator) ;
        seed_out((long *)NULL, S_evaluator) ;
}
```

If you get a message similar to the following when you run **Splus make**, you may need to add the `S_EVALUATOR` machinery to your code:

```
plum:rich[364] Splus make
cc -I${SHOME}/include  -O -Xa   -c orand.c
"orand.c", line 6: prototype mismatch: 1 arg passed, 2 expected
cc: acomp failed for orand.c
make: *** [orand.o] Error 2
```

The New .Call Function

The `.Call` function can be used to pass in and return arbitrary S-PLUS objects, including objects of user-defined classes. This gives you the freedom to work with S-PLUS objects in your C code, but also gives you much more freedom to create bugs, sometimes disastrous ones. As a simple example of how it might be used, consider the problem of computing a value whose length is determined as part of its computation. This type of computation formerly required the `POINTERS` argument to `.C`. Now it can be handled using the `.Call` interface. The following C routine takes an S-PLUS object `x` as input, and returns a sequence of length `max(x)`:

```
#include "S.h"
s_object *makeseq(s_object *sobjX)
{
    S_EVALUATOR
    long i, n, xmax, *seq, *x ;
    s_object *sobjSeq ;
```

```
/* Convert the s_objects into C data types: */
    sobjX = AS_INTEGER(sobjX) ;
    x = INTEGER_POINTER(sobjX) ;
    n = GET_LENGTH(sobjX) ;

/* Compute max value: */
    xmax = x[0] ;
    if(n > 1) {
        for(i=1; i<n; i++) {
            if(xmax < x[i]) xmax = x[i] ;
        }
    }

    if(xmax < 0)
       PROBLEM "The maximum value (%ld) is negative.", xmax
ERROR ;

/* Create a new s_object, set its length and
 * get a C integer pointer to it
 */
    sobjSeq = NEW_INTEGER(0) ;
    SET_LENGTH(sobjSeq, xmax) ;
    seq = INTEGER_POINTER(sobjSeq) ;

    for(i=0; i<xmax; i++) {
        seq[i] = i + 1 ;
    }

    return(sobjSeq) ;
}
```

You can call this code using the following S-PLUS function:

```
> "makeseq" <-
+ function(x)
+ {
+         x <- as.integer(x)
+         .Call("makeseq", x)
+ }
```

Migrating Object-Oriented Program Code

The S-PLUS object-oriented programming model has been completely revamped. The old model (creating generic functions that called `UseMethod`, and having methods with names created by concatenating the name of the generic function and the name of the class) is now deprecated. If you have old classes, you can continue to use the old model to create new methods for those old classes, but you should also create new classes and methods using the new model.

One major difference between the new and old programming models is how generic functions and methods are stored. In S-PLUS 3.x, these were ordinary functions stored in the standard system databases. In S-PLUS 5.x and later, generic functions and methods are stored as *metadata*, in special meta databases. A generic function or method stored in the metadata is used in preference to an ordinary function stored in the search path directories. If you have done much programming in S-PLUS 3.x, you will probably want to define methods via *calls* to ordinary functions, rather than by including the specific function definition in the metadata.

An example will clarify the distinction. Suppose you want to make the `chol` function the square-root method for objects of class `"matrix"`. (Remember, everything in S-PLUS 5.x and later has a class!) You can do this in two ways:

```
> setMethod("sqrt", "matrix", function(x) chol(x))
```

or

```
> setMethod("sqrt", "matrix", chol)
```

The first way defines the method via a call to `chol`, while the second defines the method by naming `chol`. In the second case, S-PLUS creates a copy of `chol` in the metadata, and will use that copy whenever the `sqrt` function is called on an object of class `"matrix"`. If you make later changes to `chol` in the ordinary database, these will not be reflected in your `sqrt` method. In the first case, however, the function stored in the metadata simply calls the function stored in the ordinary database. This allows you to store all your active functions in ordinary databases, as you did in S-PLUS 3.x.

Metadata is also very important in maintaining the inheritance structure of your old-style classes. Use `setOldClass` to specify the inheritance for your old-style classes.

In S-PLUS 3.4 and earlier, it was possible to define classes using *multiple inheritance*, where a single class could inherit from multiple, unrelated classes. For example, you might have one old class attribute defined as `c("MiVariable", "bs", "basis")` and another defined as `c("MiVariable", "factor")`. In S-PLUS 5.x and later, the second definition (as an argument to `setOldClass`) would assert that the class `MiVariable` inherits from class `factor` as well as from the class `basis`. This multiple inheritance is not supported in S-PLUS 5.X and later.

There are two possible solutions:

1. Define `MiVariable` so it has no inheritance, using a regular list object and the simple class `MiVariable`.
2. Define `MiVariable` so it inherits from a *virtual class* `MiVariableVirtual`, using a new (S-PLUS 5.x and later) class with slots.

The first solution has the advantage of flexibility: you can add components to the object and not disturb the class definition. The second solution allows you define new-style methods for your virtual class and ensure that no unwanted coercion takes place. Which approach you choose should be determined by the actual structure of the data in your class.

Updating Loops

Loops (`for`, `while`, `repeat`) no longer have return values; this was an efficiency improvement installed for S-PLUS 4.0 for Windows, but is new to the UNIX platform as of S-PLUS 5.0. In earlier versions of S-PLUS, the value of a loop was the value of the last expression in the last completed iteration of the loop. However, few S functions used this value, and we often recommended inserting `NULL`s at the end of loops to suppress this return value. Thus, the effect of this change on your code is probably negligible.

If you did make implicit use of a loop's return value, as in the following function `newton` from version 3.2 of the *S-PLUS Programmer's Manual*:

```
> newton <-
+ function(n, j = 2, x = 1)
+ {
+ # Use Newton's method to find jth root of n, starting at
+ # old.x == x
+ # Default is to find square root of n from old.x == 1
+     old.x <- x
+     repeat {
+         new.x <- old.x - ((old.x^j-n)/(j * old.x^(j-1)))
+         conv <- abs(new.x - old.x)
+         if(conv/abs(old.x) < 1e-10)
+             break
+         old.x <- new.x
+     }
+ }
```

you will need to generate an explicit return value. In this case, simply returning `old.x` will satisfy our needs:

```
> newton <-
+ function(n, j = 2, x = 1)
+ {
+ # Use Newton's method to find jth root of n, starting at
+ # old.x == x
+ # Default is to find square root of n from old.x == 1
+     old.x <- x
+     repeat {
+        new.x <- old.x - ((old.x^j-n)/(j * old.x^(j-1)))
+        conv <- abs(new.x - old.x)
+        if(conv/abs(old.x) < 1e-10)
+             break
+        old.x <- new.x
+     }
+     old.x
+ }
```

Changes in Debugging

The `inspect` interactive debugger is not available in this release of S-PLUS, and there have been several changes to the `browser` and the related function `debugger`. In addition, a new function, `recover`, can be used to provide interactive debugging as an error action. Unlike how this feature is described in *Programming with Data*, however, this is not the default.

Using recover

To use `recover`, set your error action as follows:

```
> options(error=expression(if(interactive())
+    recover() else dump.calls()))
```

Then, for those type of errors which would normally result in the message `"Problem in ... Dumped,"` you are instead asked `"Debug? Y/N"`; if you answer `"Y"`, you are put into `recover`'s interactive debugger, with a `R>` prompt. Type `?` at the `R>` prompt to see the available commands. Use `up` to move up the frame list, `down` to move down the list. As you move to each frame, `recover` provides you with a list of local variables. Just type the local variable name to see its current value. For example, here is a brief session that follows a faulty call to the `sqrt` function:

```
> sqrt(exp)

Problem in x^0.5: needed atomic data, got an object of class
"function"

Debug ?  ( y|n ): y

Browsing in frame of x^0.5
Local Variables: .Generic, .Signature, e1, e2

R> ?

Type any expression. Special commands:
`up', `down' for navigation between frames.
`where' # where are we in the function calls?
`dump'  # dump frames, end this task
`q'     # end this task, no dump
`go'    # retry the expression, with corrections made
Browsing in frame of x^0.5
Local Variables: .Generic, .Signature, e1, e2
```

```
R> up

Browsing in frame of sqrt(exp)
Local Variables: x

R(sqrt)> x

function(x)
.Internal(exp(x), "do_math", T, 108)

R(sqrt)> x<-exp(1)

R(sqrt)> go
[1] 1.648721
```

In the example session, we accidentally gave a function as the argument to `sqrt`, rather than the needed atomic data object. Inside `recover`, we move up to `sqrt`'s frame, change the argument `x` to the result of a function *call*, then use `recover`'s `go` command to complete the expression.

Using browser

The `browser` function now works much like the `recover` function: you navigate using the `up` and `down` functions, see available commands and local variables with `?`, and exit with `q`. You can insert calls to `browser` with `trace`, as in earlier versions of S-PLUS.

INDEX

Symbols

...argument 124
.First function 440
.Last function 438

A

add-on modules 3
agglomerative hierarchical method 395
aggregate fz 116
analysis of variance (ANOVA) 298, 361
    one-way 298, 302
    random effects 362
Apply button 129
argument ... 124
arguments
    abbreviating 41
ARIMA 424
Arithmetic, operators 36
as.data.frame fz 104
attach function 34, 440
autocovariance/correlation 421
autoregressive integrated moving-average (ARIMA) 424
Axes page
    in graphics dialogs 127, 136

B

bandwidth 143, 158, 417
    span 147, 151
bar chart 166
Bar Chart dialog 166
    tabulating data 168
binomial power and sample size 322, 324
Binomial Power and Sample Size dialog 322, 324
blood data 300
bootstrap 413
box kernel 144, 158
box plot 174
    for a single variable 175
    for multiple variables 176
Box Plot dialog 174
    multiple variables 176
    single variable 175
by fz 116, 120

C

calling functions 35
candlestick plot 204
cbind fz 104, 110
c function 35
character data type 123
character strings
    delimiting 35
chi-square goodness-of-fit test 285
chi-square test 271, 319
class 27
cloud plot 189
Cloud Plot dialog 189
cluster analysis
    agglomerative hierarchical 395
    compute dissimilarities 389
    divisive hierarchical 397
    fuzzy analysis 393
    k-means 390
    monothetic 399
    partitioning around medoids 392
coagulation data 299
combining data frames 109
    by column 110
    by row 112
    merging 113
    rules 123
command line editing 18

command line editor 18
command recall 20
example 19
startup 18
table of keystrokes 18
Commands window 129
compute dissimilarities 389
continuation 16
continuous response variable 299
contour plot 183
Contour Plot dialog 183
Correlations and Covariances dialog 274
cosine kernel 158
counts and proportions 308
Cox proportional hazards 376
crosstabulations 271
Crosstabulations dialog 271, 272

D

data
editing 43
importing 43
with importData function 43
reading from a file 43
data.frame data type 123
data.frame fz 104
data frames 102
adding new classes of variables 123
applying functions to subsets 116
combining objects 107
dimnames attribute 106
row names 106
rules for combining objects 123
data objects 102
combining 35
editing 44
Data Set field 130, 267
Data Viewer 128
degrees of freedom 281
delimiters
for character strings 35
density plot 158
bandwidth 158
cosine kernel 158
kernel functions 158
normal (Gaussian) kernel 158
rectangle kernel 158
triangle kernel 158
Density Plot dialog 128
divisive hierarchical method 397
dot plot 169
Dot Plot dialog 169
tabulating data 171

E

editing
command line 18
data objects 44
editing data 43
Editor 432
EDITOR environment variable 18
emacs 18
emacs_unixcom editor, table of keystrokes 18
emacs editor
table of keystrokes 18
Environment variables
PAGER 432
environment variables 433
EDITOR 18
S_CLEDITOR 18
S_CMDFILE 435
S_WORK 441
VISUAL 18
error messages 16
exact binomial test 308
examples
ANOVA of coagulation data 299
one-sample speed of light data 277
two-sample weight gain data 289

Exiting S-PLUS 15
exploratory analysis, speed of light data 278, 290
expressions
  multiple line 16

F

factor analysis 402
Factorial Design dialog 327
FASCII files
  notes on importing 88
Fisher's exact test 312
formulas 267
freedom, degrees of 281
Friedman rank test 305
FUN argument 119
functions
  calling 15, 35
  for hypothesis testing 59
  for statistical modeling 60
  for summary statistics 57
  high-level plotting 53
  importData 43
  low-level plotting 54
  operators
    comparison 37
    logical 37
    precedence hierarchy of 39
  qqnorm, for linear models 341
fuzzy analysis 393

G

Gaussian kernel 144, 158
generalized models
  linear 354
graph dialogs
  QQ Math Plot 164
graphical user interface
  Apply button 129
  Commands window 129
  Data Viewer 128
  graphics dialogs 128
  Graph menu 128
  Graph window 129
  OK button 129
  Options menu 131
  Report window 129
graphics
  dialogs for 130
  Graph menu for 128
  Graph window for 129
  Options menu for 131
graphics dialogs 128, 130
  Axes page 127, 136
  Bar Chart 166
  Box Plot 174
  Cloud Plot 189
  Contour Plot 183
  Data Set field 130
  Density Plot 128
  Dot Plot 169
  Histogram 162
  Level Plot 185
  Multipanel Conditioning page 127, 152
  Parallel Plot 194
  Pie Chart 171
  Plot page 136
  QQ Plot 180
  Scatter Plot 127, 132
  Scatter Plot Matrix 191
  Strip Plot 178
  Subset Rows field 130
  Surface Plot 187
  Time Series High-Low Plot 204
  Time Series Line Plot 200
  Titles page 127, 136
graphics examples
  barley data 196
  djia data 205
  ethanol data 153
  exsurf data 184
  fuel.frame data 167
  kyphosis data 181
  lottery.payoff data 176
  main gain data 133
  Michelson data 159
  Puromycin data 138

sensors data 144
sliced.ball data 189
graphics options 131
Graph menu 128
Graph window 129
GUI See graphical user interface

H

help.off function 21
help.start function 21
Help system
on-line help 3
training courses 4
help system 21
high-low-open-close plot See high-low plot
high-low plot 204
histogram 162
binning algorithms 163
Histogram dialog 162
hstart time series 117
hypothesis testing 58, 59

I

importData function 43
importData function 104
importing data 43
index plots 136
initialization, options function 431
interquartile range 174
interrupting evaluation 17

J

jackknife 415

K

kernel smoothers 144
box kernel 144
normal (Gaussian) kernel 144
Parzen kernel 144
triangle kernel 144
k-means method 390
Kolmogorov-Smirnov goodness-of-fit test 283, 296
Kruskal-Wallis rank sum test 303
Kruskal-Wallis Rank Sum Test dialog 304
kyphosis data frame 117

L

least squares line fits 140
in scatter plot matrices 193
level plot 185
Level Plot dialog 185
levels, experimental factor 299
linear models
diagnostic plots for 339, 340
F-statistic for 338
multiple R-squared for 338
standard error for 338
line plots 136, 200
list data type 123
list function 32
lists
components 32
loess (local) regression 348
loess smoothers 147, 418
span 147

M

make.groups function 176
MANOVA 406
Mantel-Haenszel test 317
matrix data type 123
matrix function 30
max fz 120
McNemar's test 314
mean fz 120
merge fz
by.x argument 115
by.y argument 115
merge fz 104, 113
Michaelis-Menten relationship 351
model.matrix data type 123
modeling, statistical 59, 60

modules, add-on 3
monothetic analysis 399
Multipanel Conditioning page
    in graphics dialogs 127, 152
multivariate analysis of variance (MANOVA) 406

N

Nonlinear Least Squares Regression dialog 349, 350, 352, 353
nonlinear regression 349
nonparametric curve fits 143
normal (Gaussian) kernel 144, 158
normal power and sample size 322
Normal Power and Sample Size dialog 322
numeric summaries 117

O

OK button 129
one-sample tests 276
    t-test 276
One-sample t Test dialog 276
One-sample t-Test dialog 280
One-sample Wilcoxon Test dialog 282
One-way Analysis of Variance dialog 303
on-line help 3
operators
    comparison 37
    logical 37
    precedence hierarchy of 39
Operators, arithmetic 36
Options menu 131
Orthogonal Array Design dialog 328
outlier data point 135

P

parallel plot 194
Parallel Plot dialog 194
partitioning around medoids 392
Parzen kernel 144
pie chart 171
Pie Chart dialog 171
    tabulating data 173
Plot page
    in graphics dialogs 136
plots
    bar charts 166
    box plots 174
    cloud plots 189
    contour plots 183
    density plots 158
    diagnostic, for linear models 339
    dot plots 169
    for linear models 340
    high-level functions for 53
    high-low plots 204
    histograms 162
    index plots 136
    least squares line fits 140
    level plots 185
    line plots 136, 200
    low-level functions for 54
    parallel plots 194
    pie charts 171
    qqplots 164, 180
    robust line fits 141
    scatter plot matrix 191
    scatter plots 143
    strip plots 178
    surface plots 187
    time series 200
    time series plots 204
    Trellis graphics 152, 196
    using statistics dialogs 268
precedence of operators 40
principal components technique 404
probability distributions, skewed 278
Prompts, continuation 431
Prompts, S-Plus 431
proportions parameters test 310

Q

QQ Math Plot dialog 164
QQ Plot dialog 180
qqplots 164
    normal qqplot 164
    two-dimensional 180
quantile-quantile plot See qqplots
Quitting S-PLUS 15

R

random effects analysis of variance 362
rbind fz 104, 112, 113
read.table fz 104
recalling previous commands 20
rectangle kernel See box kernel
regression 334
    linear 335
    local (loess) 348
    nonlinear 349
regression line 340
Report window 129
resampling
    bootstrap 413
    jackknife 415
residuals
    definition of 140, 335
    normal plots 341
    plotting in linear models 341
rm function 34
robust line fits 141

S

S_CLEDITOR environment variable 18
S_CMDFILE variable 435
Save As field 267
Save In field 267
Scatter Plot dialog 127, 132
scatter plot matrix 191
Scatter Plot Matrix dialog 191
    least squares line fits 193
scatter plots
    least squares line fits 140, 193
    multipanel conditioning 152
    nonparametric curve fits for 143
    robust line fits 141
    smoothers 143
    three-dimensional 189
Session options, continuation prompt 431
session options, echo 431
Session options, editor 432
Session options, printing digits 432
Session options, prompt 431
Session options, screen dimensions 432
smoothers 417
    for scatter plots 143
    kernel smoothers 144
    loess smoothers 147
    running averages 143
    spline smoothers 149
    supersmoothers 151
S-news mailing list 4
solder data set 103
span 147, 151
speed of light data 277
    exploratory analysis of 278
spline smoothers 149
    degrees of freedom 149
S-Plus 458
S-PLUS syntax
    formulae in 60
Starting S-PLUS 11, 12
starting S-PLUS 18
statistical modeling 59, 60
statistical techniques
    analysis of variance
        random effects 362
    cluster analysis
        agglomerative hierarchical 395
        compute dissimilarities 389
        divisive hierarchical 397
        fuzzy analysis 393
        k-means 390

monothetic 399
partitioning around medoids 392
comparing samples
one-sample
chi-square goodness-of-fit test 285
Kolmogorov-Smirnov goodness-of-fit test 283
t-test 276
Wilcoxon signed-rank test 281
two-sample
Kolmogorov-Smirnov goodness-of-fit test 296
t-test 288
Wilcoxon rank sum test 294
counts and proportions
chi-square test 319
exact binomial test 308
Fisher's exact test 312
Mantel-Haenszel test 317
McNemar's test 314
proportions parameters test 310
data summaries
crosstabulations 271
summary statistics 269
factor analysis 402
generalized linear models 354
k samples
Friedman rank test 305
Kruskal-Wallis rank sum test 303
one-way analysis of variance 298
multivariate analysis of variance 406
power and sample size
binomial 322, 324
normal 322
principal components 404
regression
linear 335
local (loess) 348
resampling 413
bootstrap 413
jackknife 415
smoothing
supersmoother 419
survival analysis
Cox proportional hazards 376
time series
autocovariance/correlation 421
autoregressive integrated moving-average 424
tree models 381
statistical tests
analysis of variance (ANOVA) 298, 361
one-sample 276
two-sample 287
statistics
dialogs for 266
Correlations and Covariances 274
Crosstabulations 271
Data Set field in 267
formulas in 267
Nonlinear Least Squares Regression 349, 350, 352, 353
plotting from 268
Save As field in 267
Save In field in 267
Summary Statistics 269, 279
introduction to 264
regression 334
savings results from an analysis 268
Statistics menu for 265, 266
summary 57, 269
common functions for 57
Statistics menu 265, 266

StatLib 4
strip plot 178
Strip Plot dialog 178
Student's t confidence intervals 280
Student's t significance test p-values 280
Student's t-tests 280, 291
Subset Rows field 130
summary statistics 57, 269
    common functions for 57
Summary Statistics dialog 269, 279
supersmoother 419
supersmoothers 151
    span 151
surface plot 187
Surface Plot dialog 187
survival analysis
    Cox proportional hazards 376
syntax 16
    case sensitivity 16
    continuation lines 16
    spaces 16

T

tapply fz 121
technical support 5
testing, hypothesis 58, 59
time series 200
    autocovariance/correlation 421
    autoregressive integrated moving-average 424
    candlestick plots 204
    high-low plots 204
    line plots 200
Time Series High-Low Plot dialog 204
Time Series Line Plot dialog 200
Titles page
    in graphics dialogs 127, 136
training courses 4
treatment 299
    ANOVA models 302
tree-based models 381
Trellis graphics 152, 196
    functions for 127
    panels in 153
triangle kernel 144, 158
two-sample tests 287
    t-test 288
Two-sample Wilcoxon Test dialog 295

U

unix function 42

V

variable, continuous response 299
vector arithmetic 39
vector data type 123
vectors
    creating 35
vi editor 18
    table of keystrokes 18
vi function 45
VISUAL environment variable 18

W

weight gain data 289
Wilcoxon rank sum test 294
Wilcoxon signed-rank test 281
working directory
    how set 441